Thinking Machines Corporation

Connection Machine® System Specifications

Typical Applications Performance*

General Computing	1,000 Mips
Document Search	6,000 Mips
Fluid Flow Simulation	7,000 Mips
Stereo Matching	1,000 Mips

Sorting

65,536 32-bit values	33 milliseconds

Interprocessor Communications

(32-bit messages)	
Regular Pattern	250 million/second
Random Pattern	70 million/second

Variable Word Length Arithmetic

64-bit integer add	1,150 Mips
32-bit integer add	1,980 Mips
16-bit integer add	2,600 Mips
8-bit integer add	4,600 Mips
64-bit move	1,240 Mips
32-bit move	2,350 Mips
16-bit move	4,100 Mips
8-bit move	6,600 Mips

Connection Machine System

Processors	65,536
Memory Size	32 Mbytes
Clock Rate	4 Mhz

Bandwidth

Memory	256,000 million bits/second
I/O	500 million bits/second

Physical Dimensions

Size:	$56'' \times 56'' \times 62''$
Weight:	2,600 lbs.

Environmental Requirements

(does not include host)

Power Dissipation	25 KW
Power Input	Four 30-amp 3-phase 110/208v
Operating Temperature	$70°F \pm 5°F$
Operating Relative Humidity	$50\% \pm 10\%$

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Thinking Machines Corporation has measured its product carefully. We believe that all specifications are accurate as of the date of publication. Thinking Machines Corporation can not, however, be responsible for inadvertent errors. Product specifications are subject to change without notice.

*Millions of instructions per second

®Connection Machine is a registered trademark of Thinking Machines Corporation.

Thinking Machines Corporation

45 First Street

ambridge, MA 02142-1214

17·876·1111

CORPORATE BACKGROUND

Thinking Machines Corporation was founded in May 1983 based on an understanding of how two evolving technologies -- parallel processing and artificial intelligence -- would combine to create a computer with fundamentally new and different capabilities.

The company's founders saw that the technologies for understanding information were running far behind the technologies for creating, storing, and transmitting it. They were not alone in recognizing this information overload crisis. As they were starting, the Japanese government was initiating its Fifth Generation computer project to exploit the same technologies -- artificial intelligence and parallel processing -- in a drive for world-wide industry leadership. European countries were beginning the Esprit and Alvey projects along similar lines. In America, the federal government, realizing that these developments had important strategic implications for the balance of world power, launched a major program within the Defense Advanced Research Project Agency (DARPA).

Thinking Machines held three key advantages in this competition for the next generation of computing capability. First, the founders had a clear idea of the functions required of

(more)

a true breakthrough computer. Second, they had access to the very best artificial intelligence technology. And, third, they had a clear belief that a small organization would yield faster and more useful results than the large development efforts under way in other parts of the world.

At the outset, Thinking Machines President and principal founder, Sheryl Handler, began assembling an exceptional team of artificial intelligence experts, computer scientists, and senior managers in a unique, interdisciplinary approach to the field. These included W. Daniel Hillis, Founding Scientist and acknowledged leader in massively parallel system design; Marvin Minsky, Founding Scientist, co-founder of the Massachusetts Institute of Technology Artificial Intelligence Laboratory, and internationally recognized authority on artificial intelligence research and applications; Richard J. Clayton, Vice President for Product Development, former Vice President for Computer Systems Development at Digital Equipment Corporation; Marvin Denicoff, Chief of Project Development and former architect of the U.S. government's programs in artificial intelligence and related research; and Mirza Mehdi, Vice President for Corporate Development, whose experience combines finance and strategic planning at large multi-national and emerging high technology companies. Early investors included William S. Paley, Frank Stanton, and Benno Schmidt.

(more)

Additionally, Handler formed a distinguished group of "corporate fellows" -- world leaders in artificial intelligence, physics, mathematics and computer science -- to participate in the company's pioneering research. The first two fellows were Nobel Laureate Richard Feynman, professor of theoretical physics at the California Institute of Technology; and Jerome Wiesner, President Emeritus and Institute Professor at MIT.

During subsequent months, the company applied these multiple disciplines to the development of hardware and software advances that would amount to nothing less than a new generation of computing. Because of the global importance of the project, Thinking Machines continued to attract some of the world's leading computer scientists, including David Waltz from the University of Illinois, Rolf D. Fiebrich from International Business Machines Corp., and Guy Steele from Carnegie-Mellon University. As well, it built the financial, manufacturing, and marketing resources needed to bring this technology to commercial application.

In November 1985, following two years of research and development, Thinking Machines delivered its first computer system, the Connection Machine® computer, to the Defense Advanced Research Project Agency (DARPA), which traditionally has funded some of the most significant developments in computer technology.

(more)

Digitized by the Internet Archive
in 2026 with funding from
Brewster Kahle

https://archive.org/details/thinkingmachines00unse_6

Subsequently, on April 30, 1986, the company introduced the first commercial version of the Connection Machine system, a massively parallel computer that can process extremely large volumes of data at speeds exceeding one billion instructions per second (1,000 MIPS).

"The demand for a computer with fundamentally new capabilities has never been in question," says Handler. "We were determined to be the first to supply it."

Such computational power is possible because of the Connection Machine system's unique parallel architecture, which differs substantially from other computer systems. While some machines speed a problem's time to solution by linking a small number of processors in parallel -- from a half dozen to several hundred -- the Connection Machine system's massively parallel design employs up to 64,000 individual processors, each with its own memory, that compute simultaneously. Unlike other parallel computers, whose processors are pre-wired to communicate in a specific pattern, the Connection Machine processors are automatically linked by the machine to match the structure of each problem as it is being solved.

The Connection Machine System

Thinking Machines recognized from the outset that business and government needed a computer able to handle data at a rate considered impossible using conventional technology. The

(more)

system had to understand huge amounts of unstructured data, not just create more of it. It had to compute directly at the level of the data itself.

These requirements pointed the way to a radical new architecture, with a very large number of processors embedded directly in the memory, where the data resides. This architecture addresses the inherent parallelism that exists at the data level of many important problems. At the data level it is common to do thousands of operations in parallel. Conventional multiprocessor computers rarely achieve speedups of more than ten, and even these incremental advances require more complex programming techniques.

The crucial design challenge for Thinking Machines lay in the way these processors were interconnected. The company's scientists, led by Hillis, had to make the connections fast, but they also had to make them flexible. Systems with a rigid network of connections limit themselves to a small set of applications. However, a system that allows free and easy connections among tens of thousands of processors can provide exactly the kind of computing power the world needs as it moves into the "information economy" of the 21st century.

The Connection Machine system is the dramatic result of this design effort. Its 64,000 processors operate at sustained speeds above 1,000 MIPS. (In contrast, large conventional mainframes operate at less than 40 MIPS.) The Connection Machine processors

are linked by a communications system that exchanges data at speeds above 3 billion bits per second, a rate equivalent to 1,000 conventional computer networks. Linkages between processors are totally dynamic; they change to match the changing nature of applications programs.

For users, the result is a computer that looks at the whole problem at once. Its 64,000 processors search whole databases, process whole pictorial images, and simulate whole VLSI circuits simultaneously. It is a much simpler approach that is also a much higher performance approach. The parallelism inherent in data structures grows with the size of the data. Processing all the picture elements in an image, for example, is as fast as processing one of them. A whole database can be searched in the time it takes to search one document.

The Market

Initial Connection Machine applications have provided proof that the design is right. In word and language applications the system is allowing certain artificial intelligence algorithms to be applied to real world problems for the first time. Among these algorithms is a method for searching huge unstructured databases through whole document comparison. In picture and vision applications, the system is allowing image comparison algorithms to operate in seconds instead of minutes or hours.

(more)

One example is the computing of contour maps from aerial photographs taken from slightly different angles.

Numeric and scientific applications are providing some of the earliest uses for the Connection Machine system. To remain true to the physical reality, mathematical models must often be "non-uniform," more dense in one place, less dense in another. Conventional vector computers force a uniform structure onto these models. The Connection Machine system adapts directly to the natural structure of the problem. Fluid dynamics simulation and linear programming are just two examples of scientific and numeric applications of the Connection Machine system.

The company has focused its initial product on large-scale users in industry, universities, and government agencies. These customers have already invested in large-scale computing resources, but even the largest conventional machines are inadequate for their needs. They are looking to a new technology to break their information understanding bottleneck. For example, because the Connection Machine system is well suited to reducing the time to search large quantities of unstructured text, information services organizations are prime potential customers.

The Future

How will Thinking Machines maintain its lead? Handler points to the company's inherent strengths: "Thinking Machines

is a combination of compelling vision and excruciatingly high standards for execution. It is my job to make progress without compromising either."

She describes the company's approach: "We seek a blend of the finest minds. We create fire by rubbing them together in an environment of sharp focus and superb tools. It gets results."

The company sees the Connection Machine system as the building block of a major new industry. "The architecture is applicable to a wide range of problems, and it's extremely cost-effective," notes Clayton, who is responsible for Connection Machine business operations. "We are growing to meet demand."

Much of the growth is expected to come from new applications. The company sees its architecture as an enabling technology for information understanding. Systems that understand unstructured information will be possible for the first time, thereby opening up new markets.

Perhaps most important, the company sees the Connection Machine system as a key milestone on the road to a true thinking machine. "Every time we move a step closer to that goal," notes Hillis, "we gain power to understand the flood of information around us. This fills a real need, and filling a real need is what makes a company successful."

# # # #

Connection Machine® is a registered trademark of Thinking Machines Corp.

For further information contact: James Bailey
Thinking Machines Corp.
(617) 876-1111

or

Maura FitzGerald
Miller Communications
(617) 536-0470

Thinking Machines Corporation

245 First Street

Cambridge, MA 02142-1214

617-876-1111

TECHNOLOGY BACKGROUND

The technologies for creating, storing, and transmitting information have moved ahead rapidly in the past two decades. The technologies for understanding that information have lagged far behind. For some time, it has been clear to the scientists at Thinking Machines that a real solution to this problem required nothing less than a fundamentally new approach to computing: an approach optimized for understanding information, not just creating more of it. This new technology is called "data-level parallelism" and it forms the heart of the company's Connection Machine computer.

Like other forms of parallelism, data-level parallelism gains efficiency by doing many things at once. There are two major sections to any computer application, the control section (the program instructions) and the data section. Large applications have tens of thousands, or even millions, of data elements. Many of the instructions in the control sequence are independent; they may in fact be executed in parallel by multiple processors. This is known as "control level parallelism", the technique used by "multi-processor" computer systems. In the same way, many of the data elements are independent; operations on these data elements may be carried out simultaneously by multiple processors. This is data-level parallelism.

(more)

Thinking Machines architects focused at the data level because it holds such enormous potential for maximizing performance. With control level parallelism, it is rare to be able to do more than ten to fifteen operations in parallel. With data-level parallelism, it is common to do thousands of operations in parallel.

Origins of Data-level Parallelism

Data level parallelism is inherent in the problem itself. The larger and more complicated a problem's data structure, the more parallelism there tends to be. For example, consider the simulation of a VLSI circuit. (VLSI manufacturers simulate their circuit designs on a computer before committing to an expensive production run.)

The simulation consists of modelling the behavior of each transistor in the circuit. The data for the problem consists of the current status (voltage, current, charge, and conductance) of each transistor. The control sequence consists of the computations necessary to update a transistor's status at each time step of the simulation.

At the data level, the amount of parallelism is enormous, because the operations that are being done on one transistor can be done on all the others (typically thousands) at once. The same inherent parallelism exists in vision applications,

dictionary searches, and many other problems. As Professor Oliver McBryan of the Courant Institute notes, "A wide range of simulations for physical processes involve high degrees of parallelism. For example, some processes involved in weather prediction may be organized into as many as a million or more parallel computations. Similar remarks apply to oil reservoir simulation, finite element structural analysis, aerodynamic calculations, and combustion physics."

The Connection Machine System

The Connection Machine system provides a direct and powerful implementation of data level parallelism. Individual processors are assigned to individual elements of data, allowing operations on all data elements to proceed in parallel. The system literally looks at the whole problem at once.

If there are fewer than 64,000 data elements in the problem, the system assigns a complete processor to each one. If there are more than 64,000, the system operates in virtual processor mode, operating on data sets with as many as a million elements. The power of 64,000 processors all operating in parallel yields performance above 1,000 MIPS in a wide range of application areas. By contrast, the biggest conventional large-scale computers operate at less than 40 MIPS.

(more)

Importance of Interconnections

Large numbers of processors represent only part of the technology needed for true data-level parallelism. Computing at the data level also requires matching the interconnections between data elements. Indeed, without general communications, parallel computers have limited usefulness. The VLSI circuit application demonstrates the point. Transistors in a circuit operate in parallel, but not in isolation. The wires in the circuit tie the transistors together. A data-level computer simulation must do the same with its processors.

The Connection Machine system allows processors to make connections dynamically. In this way, the system exactly matches the relationships of the data it is computing. The linkages between processors can change from one program to the next or dynamically within stages of a single program. Information is passed between processors at a rate of 3 billion bits per second, a thousand times the speed of a conventional computer network.

Simplicity of Programming

Looking at the whole problem at once is a more natural approach to computing. It results in simpler programming, because there is no need to break up a program and assign

(more)

sections of it to processors in order to get the benefits of parallelism. The processors are not assigned to the instruction sequence; they are assigned to the data.

Moreover, a single copy of the instruction sequence is used by all the processors. In the VLSI example, the instruction sequence tells each processor how to simulate a transistor. The program is stored in a front end computer (either a Digital Equipment Corp. VAX® or a Symbolics 3600®) and broadcast, an instruction at a time, to all processors, which execute it in parallel on their own unique data. The instruction sequence also includes commands to pass data to other processors as appropriate. There is no need for individual processors to store copies of the program; the copy stored on the front end computer serves for all.

The Future Technology Path

Applications running on the Connection Machine system today show that a computer designed specifically to work at the data level can solve problems beyond the scope of conventional computers. But what about tomorrow, when the problem of processing and understanding information will be even more intense?

The great advantage of data-level parallelism is that it scales up as the need for greater computing power grows. The Connection Machine architecture is easily expanded to the million processor level and beyond. Thinking Machines Corporation is

Thinking Machines Corporation

First Street
nbridge, MA 02142-1214
·876·1111

CONNECTION MACHINE APPLICATIONS

The Connection Machine® system's combination of power and flexibility at the data level make it appropriate for a wide range of problems that have been difficult or impossible to solve with traditional computer technology.

The system's key advantage is that it looks at whole problems at once. Its 64,000 processors search entire databases, compute across complete visual images, and simulate complete electronic circuits simultaneously, at speeds in excess of one billion instructions per second (1,000 MIPS).

The system can adapt equally well to the data patterns of word applications, visual applications, and numerical applications. Connections between processors are dynamic, changing as needed to match the problem. Data formats are dynamic as well, from one bit per word to thousands of bits per word, depending on the needs of the problem.

This combination of processing power and data flexibility makes the Connection Machine system ideal for problems with large and complex data structures. As the following examples show, these problems exist in many categories of applications.

(more)

Example Applications: Words

Retrieval of unstructured text is one of the most difficult applications in word and language computing, and the Connection Machine system is revolutionizing the field of unstructured text retrieval. In an application where new users typically require a full day of training, the Connection Machine Document Retrieval System requires 10 minutes of training. Instead of requiring a staff of indexers to categorize incoming text, the system performs this function automatically. Where conventional systems often fail to deliver even 20 percent of the documents the user really needs to see, the Connection Machine system is achieving recall rates of 80 percent and more.

The key to this application is the program's whole document search algorithm. No intricate "Boolean expressions" are needed to refine a search. Once one or two documents are found, the user simply points to them and instructs the system to "find all the other documents on the same subject." The system extracts all the important, content-bearing terms from the example and compares them to other whole documents.

It took the combination of artificial intelligence and parallel processing technologies to solve this problem. Artificial intelligence algorithms allow the important, content-bearing terms to be extracted from documents automatically. They also allow these terms to be weighted

according to their actual importance, so the search is accurately focused. Parallel processing allows these terms to be compared to the contents of tens of thousands of other documents in a fraction of a second. This is because the system automatically assigns an individual processor to each document, allowing the system to search a whole database at once. The benefit to users is immediate: they ask their questions in a more natural way and they get more focused and complete answers faster than ever before.

Example Applications: Images

The Connection Machine system is a perfect match for a wide range of image processing applications. For example, it dramatically increases the speed of visual information processing, frequently at rates 1,000 times as fast as conventional computers.

Image display is critical to applications where complex information must be digested readily by a human expert. As Professor Nicholas Negroponte, Director of the M.I.T. Media Arts Laboratory and an early Connection Machine customer, notes, "We are developing a very fast way to produce holograms from computer images in parallel. This is very important for medicine. We know from tests that doctors can find an anomaly in a true 3-D image much more effectively than when they use current display techniques, and holograms can give that true 3-D representation using data from CAT scanners and NMR devices."

(more)

The Connection Machine system is also being used to process huge volumes of incoming images. In a mapping application, contour maps are computed and displayed automatically from two overhead images taken from slightly different angles. Because it looks at the whole problem at once, the Connection Machine system is able to produce contour maps at a rate of 7,000 per hour, compared to 15 per hour on a conventional machine.

Example Applications: Numbers

Numeric applications provide some of the most challenging examples of unstructured data. To accurately capture physical reality, mathematical models must often be non-uniform, more dense in one area and less dense in another. As Professor Oliver McBryan of the Courant Institute notes, "Problems such as those from structural analysis or from vortex simulations, where complex and irregular communications are required between distant processors, will benefit most from the high connectivity of the Connection Machine communications network."

VLSI simulation is another example of a numeric problem with complex and irregular communications. "Users of conventional computers rarely try to simulate more than 50 to 500 transistors at a detailed level," notes Rolf D. Fiebrich, director of the design automation group at Thinking Machines. The Connection Machine system is performing full, detailed simulations on circuits as

large as 8,000 transistors, with results available in 15 minutes or less.

Aggregate Behavior

Powerful though they are, words, images, and numbers are simply abstractions of the world around us. They simplify information and behavior that are too complicated to deal with directly. Now, with the Connection Machine system, users are reaching and modeling that behavior directly.

Fluid dynamics simulation is an excellent example. Computer models of fluids form the basis of modern weather forecasting, airplane design, automobile streamlining, and ship-hull contouring. While traditional simulations must employ complicated and imperfect mathematical formulas, the Connection Machine system models the movement and jostling of individual molecule groups directly. As billions of individual interactions occur, the fluid behavior emerges.

Moreover, because the Connection Machine system makes fluid simulation an interactive process, a user can now alter and model an object in minutes instead of hours.

The Future

The Connection Machine system's ability to operate on words, images, and numbers offers, for the first time, the ability to

utilize all three processing modes in a single application. This means that users performing numerical simulations will be able to draw on knowledge bases of unstructured information and display their results in more sophisticated visual images. It makes possible image processing applications that will incorporate reasoning capabilities. In short, a whole new level of applications capability has been opened, and it is this capability that will form the basis of world leadership in the race to create the next generation of computer technology.

# # # #

Connection Machine[R] is a registered trademark of Thinking Machines Corporation.

For further information, contact: James Bailey
Thinking Machines Corp.
(617) 786-1111

or

Maura FitzGerald
Miller Communications
(617) 536-0470

Thinking Machines Corporation
Technical Report Series

Thinking Machines Technical Report 86.14

Introduction to Data Level Parallelism

With Programming Examples
for the Connection Machine® System

April 1986

© 1986 Thinking Machines Corporation

"Connection Machine" is a registered trademark of Thinking Machines Corporation.
"C*" and "*Lisp" are trademarks of Thinking Machines Corporation.

Contents

1 Data Level Parallelism **1**

1.1 Parallelism in the World Around Us 1
1.2 Parallelism in Computer Systems 1
1.3 Two Styles of Computer Parallelism 2
1.4 The Connection Machine Data Level Parallel Computer 2
1.4.1 Program Execution 3
1.4.2 The Connection Machine Processors 3
1.4.3 Connection Machine I/O 4
1.5 Communications: The Key to Data Level Parallelism 4
1.6 Connection Machine Application Examples 5

2 Document Retrieval **7**

2.1 Accessing Computer Data Bases 7
2.2 Algorithms for Document Retrieval 8
2.3 Database Loading on the Connection Machine System 8
2.4 Document Lookup on the Connection Machine System 11
2.5 Retrieving the Highest Scoring Documents 12
2.6 Timing and Performance 13
2.7 Summary and Implications 14

3 Fluid Dynamics **15**

3.1 The Method of Discrete Simulation 16
3.2 A Discrete Simulation of Fluid Flow 16
3.3 Implementation on the Connection Machine System 18
3.4 Interactive Interface 21
3.5 Timing and Performance 23
3.6 Summary and Implications 23

4 Contour Maps from Stereo Images **25**
4.1 Analyzing Aerial Images by Computer 25
4.2 Seeing in Stereo 26
4.3 Finding the Same Object in Both Images 27
4.4 Matching Edges 29
4.5 Measuring Alignment Quality 29
4.6 Drawing Contour Maps 31
4.7 Finding Edges on the Connection Machine System 32
4.8 Matching Edges on the Connection Machine System 33
4.9 Drawing Contours on the Connection Machine System 36
4.10 Timing and Performance 38
4.11 Summary and Implications 38

5 The C* Programming Language **39**
5.1 C* Extensions 39
5.1.1 Parallel Control Flow 40
5.1.2 The Selection Statement 41
5.1.3 Computation of Parallel Expressions 41
5.1.4 Data Movement 43
5.2 Summary 43

6 The *Lisp Programming Language **45**
6.1 Fundamentals of Lisp 45
6.1.1 Lisp Functions 46
6.1.2 Variables 46
6.1.3 Program Control Structure 47
6.2 *Lisp Extensions 47
6.2.1 Processors 47
6.2.2 Parallel Variables 48
6.2.3 Accessing Pvars Relative to a Grid 50
6.2.4 Selection 50
6.2.5 *Lisp Programs 50
6.3 Summary 50

7 The Connection Machine System **51**
7.1 Connection Machine Internal Structure 51
7.2 Connection Machine Instruction Flow 52
7.3 Computational and Global Instructions 53
7.4 Communications Instructions 53

7.5 The Routing Process 55
7.6 Dynamic Reconfiguration 56

8 Looking to the Future **57**

List of Figures

2.1 *Documents on the same subject have a high overlap of vocabulary.* 9
2.2 *Documents on different subjects have low overlap of vocabulary.* 9

3.1 *Unless particles are obstructed by an obstacle, or collide into other particles, they continue in the same direction.* 17
3.2 *Situations that cause particles to change directions.* 18
3.3 *Hexagonal cells with six incoming bits for particle direction and six outgoing bits for particle direction* 19
3.4 *The formation of a fluid flow phenomenon, called a "vortex street," as fluid flows from left to right past a flat plate.* 22

4.1 *An oblique view of a terrain model used in a demonstration of the contour mapping algorithm.* 27
4.2 *A stereo pair of the terrain in Figure 4.1, obtained from directly above the terrain.* 28
4.3 *An example of edges. These edges were derived from the stereo pair shown in Figure 4.2. They delineate the boundaries between areas of different intensity.* 28
4.4 *An illustration of the sliding process. Each of these images shows the contents of an alignment-table-slot in each pixel. The Nth image shows slot N in every pixel's alignment table. The dark areas are regions of good alignment, i.e., areas where the same alignment-table-slot is filled in many pixels.* 30
4.5 *A contour map of the terrain model shown in Figures 4.1 and 2, computed on the Connection Machine system.* 32

Chapter 1

Data Level Parallelism

1.1 Parallelism in the World Around Us

Whenever many things happen at once, parallelism is at work. It is at work for one of two reasons: either because someone is in a hurry or because it is the natural course of events. If, for example, many people are working at once to compose a song, it is because someone is in a hurry. Music is a naturally sequential process. Physical phenomena, on the other hand, are almost always parallel. The wind in a wind tunnel does not blow over one square centimeter of an automobile body at a time. It blows across the whole frame at once, showing the engineers how the flow in one section interacts with the flow in another. If we simulate the wind in parallel, the results come faster as a natural consequence. The parallelism is being utilized, but it is not being artificially imposed. Other examples of fundamentally parallel phenomena include vision processing, information retrieval, and many types of mathematical operations.

1.2 Parallelism in Computer Systems

The same two motivations, doing things in a hurry and doing things more naturally, also motivate computer architects. Until recently, those architects who are focused on greater speed have obtained it from faster circuitry. Making the electronics twice as fast, or the memory twice as big, has traditionally been a cost-effective way to double the performance of a single-processor computer system. But now these gains have become much harder to achieve. Limits to circuit speed have been reached. So designers who are solely focused on speed are now seeking to inject parallelism into their designs. If two computers of traditional architecture can operate in parallel, the overall speed of the system can double.

There is, however, another starting point for the design process. Computer architects

can go back to the problems themselves and understand the parallelism that has been there all along. Having understood it, they can build a system that exploits it directly. The first benefit of this approach is simplicity. A computer that fits the problems it solves is easier to use and program than a computer that doesn't. And it is also faster. Systems that couple to the inherent structure of a problem mine a deeper vein of parallelism. For this reason, they can dramatically outperform systems whose superficial performance specifications seem superior. When parallelism is imposed on a problem, a speed-up of ten is considered good. When inherent parallelism is exploited, speed-ups of 1000 are commonplace.

Some applications benefit much more than others. While certain problems do not have a large amount of parallelism, there is a large and growing body of important problems that do. For these applications the method of designing the computer around the inherent parallelism of the problem is proving to be outstandingly valuable. This approach is called "data level parallelism." The remaining sections of this report describe data level parallelism and its application to three very different computing problems. The implementation examples use the Connection Machine system, the first data level parallel computer available on the commercial market. (See reference [8] for further discussion of the Connection Machine system)

1.3 Two Styles of Computer Parallelism

All computer programs consist of a sequence of instructions (the control sequence) and a sequence of data elements. Large programs have tens of thousands of instructions operating on tens of thousands, or even millions of data elements. Parallelism exists in both places. Many of the instructions in the control sequence are independent; they may in fact be executed in parallel by multiple processors. This approach is called "control level parallelism." On the other hand, large numbers of the data elements are also independent; operations on these data elements may be carried out in parallel by multiple processors. This approach, as mentioned in the previous section, is called "data level parallelism." Each approach has its strengths and limitations. In particular, data level parallelism works best on problems with large amounts of data. Small data structures generally do not have enough inherent parallelism at the data level. When the ratio of program to data is high, it is often more efficient to use control level parallelism. But control level parallelism requires the user to break up the program and then maintain control and synchronization of the pieces.

1.4 The Connection Machine Data Level Parallel Computer

The Connection Machine computer from Thinking Machines Corporation is the first system to implement data level parallelism in a general purpose way. Since the computer is designed

around the structure of real world problems, the best way to understand the Connection Machine architecture is to follow its use in solving an actual problem. A VLSI simulation example will be used for that purpose. In VLSI simulation, the computer is used to verify a circuit design before it is released to be manufactured. The Connection Machine system provides a very direct way to perform this simulation. Each transistor in the circuit is simulated by an individual processor in the system. The chapters which follow explain three more examples in much greater detail.

1.4.1 Program Execution

Data level parallelism uses a single control sequence, or program, and executes it one step at a time, just as it is done on a traditional computer. The Connection Machine system utilizes a standard architecture front end computer for this purpose. All programs are stored on the front end machine. Its operating system supports program development, networking, and low speed I/O. The front end computer has access to all the memory in the system, albeit one data element at a time because it is a serial computer.

All Connection Machine program execution is controlled by the front end system. A Connection Machine program has two kinds of instructions in it: those that operate on one data element and those that operate on a whole data set at once. Any single-data-element instructions are executed directly by the front end; that is what it is good at. The important instructions, those that operate on the whole data set at once, are passed to the Connection Machine hardware for execution.

In the VLSI simulation example, the important instructions are the ones which tell each processor to step through its individual transistor simulation process. Each processor executes the same sequence of instructions, but applies them to its own data, the data that describes the voltage, current, conductance, and charge of its transistor at that time step of the simulation.

1.4.2 The Connection Machine Processors

In order to operate on the whole data set at once, the Connection Machine system has a distinct processor for each data element. The system implements a network of 65,536 individual computers, each with its own 4096 bits of memory. The data that describe the problem are stored in the individual processors' memories. During program execution, whenever the front end encounters an instruction which applies to all the data at once, it passes the instruction across an interface to the Connection Machine hardware. The instruction is broadcast to all 65,536 processors, which execute it in parallel.

Applications problems need not have exactly 65,536 data items. If there are fewer, the system temporarily switches off the processors that are not needed. If there are more problem elements, the Connection Machine hardware operates in virtual processor mode.

Each physical processor simulates multiple processors, each with a smaller memory. Virtual processing is a standard, and transparent, feature of the system. A Connection Machine system can easily support up to a million virtual processors. In general, a problem should have between ten thousand and a million data elements to be appropriate for the Connection Machine system.

1.4.3 Connection Machine I/O

Since the front end system has access to all Connection Machine memory, it can load data into that memory and read it back out again. For small amounts of data, this is a practical approach, but for large amounts it is too slow. A separate 500-megabit-per-second I/O bus is used instead. This bus is used for disk swapping, image transfer, and other operations which exceed the capacity of the front end.

1.5 Communications: The Key to Data Level Parallelism

Large numbers of individual processors are necessary for data level parallelism, but by themselves they are not enough. After all, there is more to a VLSI circuit than individual transistors. A circuit is made up of transistors connected by wires. Similarly, there is more to a Connection Machine system than just processors. A Connection Machine system is made up of processors interconnected by a massive inter-connection system called the router.

The router allows any processor to establish a link to any other processor. In the case of the VLSI simulation example, the links between processors exactly match the wiring pattern between the transistors. Each processor computes the state of an individual transistor and communicates that state to the other processors (transistors) it is connected to. All Connection Machine processors may send and receive messages simultaneously. The router has an overall capacity of three billion bits per second.

It is part of the reality of the world we live in that many things happen at once, in parallel. It is part of the beauty of the world we live in that these many things connect and interact in a variety of patterns. Looking at the whole problem at once requires a computer that combines the ability to operate in parallel with the ability to interconnect.

Since the structure of each problem is different, the interconnection pattern of the computer must be flexible. All linkages between Connection Machine processors are established in software. Therefore, the system can configure its processors in a rectangular grid for one problem and then into a semantic network for the next. Rings, trees, and butterflies are other commonly used topologies. The chapter on hardware describes router operation in greater detail.

1.6 Connection Machine Application Examples

Each of chapters 2, 3, and 4 describes a Connection Machine example in detail. First the algorithm is described, and then the actual program that implements this algorithm is presented and discussed. It is not necessary to study the program to appreciate the simplicity of the overall approach. Many readers will want to skip over these details. The third example, contour mapping, is quite sophisticated. Hence the program for this example is more complex than the two that precede it.

The initial Connection Machine languages are C* and *Lisp. C* is an extension of C and is appropriate for a wide range of general purpose applications. *Lisp is an extension of Lisp. Lisp, while less well known than C, is also an appropriate language for a wide variety of applications. Its primary use, however, has been in the field of artificial intelligence. Chapters 5 and 6 provide an introduction to these languages.

Chapter 2

Document Retrieval

There is too much to read. The written material for almost every discipline grows much faster than any one person can read it. Computers have not provided much relief to date. Now data level parallelism provides the computing power to implement significantly better solutions to the document retrieval problem. These solutions are more natural, so they require less user training. And they are much more accurate, so they give the user much greater confidence in the results.

2.1 Accessing Computer Data Bases

There are a number of systems today that provide on-line access to text information, but they perform poorly because they rely on a "keyword" mechanism for finding documents. The premise of a keyword system is that the relevance of a whole document can be determined by the presence or absence of a few individual words. Users enter one or more "keywords" or labels that they feel capture the sense of the information needed. All documents which either contain these words or have been indexed under these words are retrieved. Those that do not are ignored. Even with refinements, such as "Find all occurrences of 'New England Patriots' within ten words of 'Superbowl'," a keyword search generally tends to either find too many documents for the user to cope with, or too few for the user to find useful. It is a guessing game, with the user trying to imagine the most fruitful search terms.

Not all relevant documents contain the one particular word that the user chose, because writers use language differently. A search for documents containing the word "chips" may find five relevant documents, but miss ten others that were indexed under "integrated circuits" or "VLSI." Since the search yields only one third of the relevant documents, it would be considered to have a *recall* of 33%. Worse yet, the five relevant documents might be returned mixed into twenty other documents describing cookies or paint or other subjects

where the word "chips" appears. Such a search would be considered to have a *precision* of 20%. Recent published testing has shown that recall results of as little as 20% are common with keyword based systems [1].

In short, keyword-based systems are very good at finding one or two relevant documents quickly. What they are poor at is producing a refined result with high recall and high precision. The Connection Machine document retrieval system provides a very powerful way for doing complete searches. It starts out using a keyword approach, but once the first relevant document is found, the whole approach changes. The user proceeds by simply pointing to one or more relevant documents and saying, in effect, "Find me all the documents in the database that are on the same subjects as this one." A document that has been identified as relevant by the user is referred to here as a "good document."

2.2 Algorithms for Document Retrieval

Data level parallelism makes massive document comparisons simple. The basic idea is this: a database of documents is stored in the Connection Machine system, one or more documents per processor. Once the first good document is found, it is used to form a search pattern. The search pattern contains all the content words of the document. The host machine broadcasts the words in the pattern to all the processors at once. Each processor checks to see if its document has the word. If it does, it increases the score for its document. When the entire pattern has been broadcast, the document that most closely matches the pattern will have the highest score, and can be presented first to the user.

The algorithm is simple to program because it takes advantage of innate characteristics of documents rather than programming tricks and second guessing. Every document is, in effect, a thesaurus of its subject matter. A high percentage of the synonyms of each topic appear because writers work to avoid repetition. In addition, variants of each word (such as plural, singular, and possessive forms), and semantically related terms also appear among the words in a particular article. Clearly not every synonym, variant, and related term will occur in a single article, but many terms will. Each reinforces the connection between the search pattern and the document. Spurious documents, on the other hand, will not be reinforced. The word "chip" will appear in an article about cookies, but "VLSI" and "integrated circuit" simply will not. In the overall scoring, truly useful documents are reliably separated from random matches. (See figures 2.1 and 2.2.)

2.3 Database Loading on the Connection Machine System

A document database may be constructed from sources of text such as wire services, electronic mail, and other electronic databases. For this description it is important to draw a

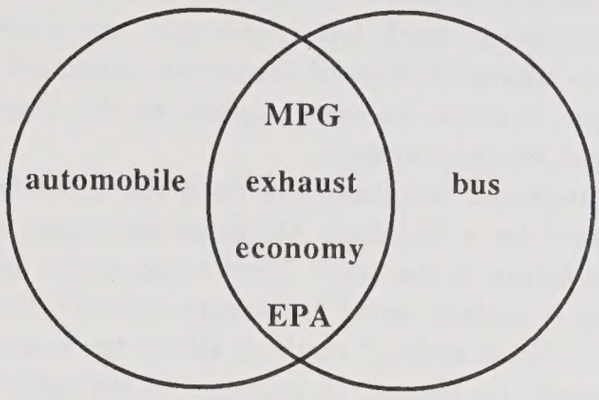


Figure 2.1: *Documents on the same subject have a high overlap of vocabulary.*

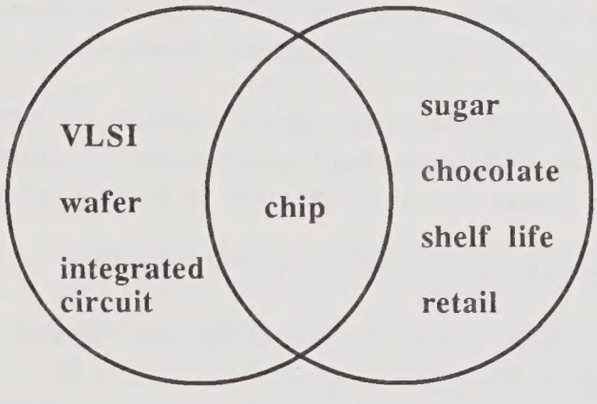


Figure 2.2: *Documents on different subjects have low overlap of vocabulary.*

distinction between *source documents* and *content kernels.* A *source document* contains the full actual text of a particular article, book, letter, or report, and is stored on the front-end's disk. A *content kernel* is a compressed form of the source document that encodes just the important words and phrases. It omits the commonplace words. Content kernels are stored in the memory of Connection Machine system.

The content kernel is produced automatically from the source document. First, the source document is processed by a Thinking Machines document indexer program that marks the most significant terms in the text. Next these terms are encoded into a bit-vector data structure, using a method called "surrogate coding." Surrogate coding, which is sometimes referred to as a "hash coding" method, allows the content kernel to be stored more compactly. It also speeds up the search process. In surrogate coding, each term in the content kernel is mapped into ten different bits in a 1024-bit vector. The ten selected bits in the vector are set to one to indicate the presence of the word in the document. In a content kernel of 30 terms, the process of surrogate coding ends up marking about a third of the bits as ones.

The source document in its original form is available for retrieval and presentation to the user when needed. The location of the original document on the system disk is stored with the content kernel.

Each segment of the content kernel is made up of the following fields:

`*score*` is used by the document lookup program to accumulate the ranking of each content kernel in the database according to how closely the content kernel matches the user's search pattern. Each time a match is found, `*score*` is updated.

`*document-id*` contains a reference to the original source document that this content kernel was derived from. When a content kernel is selected from the database lookup, the user is shown the source document referred to by this index.

`*kernel*` is a table of the surrogate-coded bit-vector encoding.

The necessary declarations for these fields are as follows. (In this chapter only, all of the code is presented twice, first in the *Lisp language and then in the C* language, to make it easy to compare the two languages. Because the characters * and ? may not appear in C* identifiers, such *Lisp names as `*score*` and `word-appears?` are rendered in C* simply as `score` and `word_appears`.)

```
;;; Declarations for the *Lisp version.

(defconstant table-size 1024)
(defconstant hash-size 10)
```

```
(*defvar *score*)
(*defvar *document-id*)
(*defvar *kernel*)


/* Declarations for the C* version. */

#define TABLE_SIZE 1024
#define HASH_SIZE 10

poly unsigned score, document_id;

poly bit kernel[TABLE_SIZE];
```

2.4 Document Lookup on the Connection Machine System

During the first stage of document lookup, the user lists a set of terms to be used to search the database, and receives back an ordered list of documents that contain all or some of those terms. The user then points to a document which is relevant, and from this document an overall *search pattern* of content-bearing words is assembled. The search pattern is simply a list of these words, with weights assigned to each word. The weight assigned to a word is inversely proportional to its frequency in the database (for example, "platinum" appears in the database less frequently than "gold," and therefore has a higher weight associated with it). This weighting mechanism ensures that uncommon words have more of an influence than common words over which content kernels get selected during the document lookup process.

Next, the search pattern is broadcast to all processors in the Connection Machine system. The same mechanism that is used to code each word in the content kernel as a series of bits is applied to the words in the search pattern. For each word in the search pattern a set of ten bit indices is broadcast. All content kernels that have these same ten bits set will have the weight of that word added into their `*score*` field. (It is possible that all ten bits for a word might happen to be set on account of other words even though that word doesn't really appear in the source document. Such an accident will result in a "false hit" on that word. However, for two reasons, this will not seriously affect the results of the lookup. First, the probability of a false hit is small: $(\frac{1}{3})^{10}$, or less than one in 50,000. Second, a false hit will be only one of many terms contributing to the score, and so will have only a small effect even when it does occur.)

The following code is used to broadcast one search pattern word to all the processors

in the system, which check their content kernels and add the value of weight into their *score* if it contains the word. The word is represented by a list of ten bit locations (bit-locs).

```
;;; *Lisp code for testing the presence of a single word.

(*defun increment-score-if-word-appears (bit-locs word-weight)
  (*let ((word-appears? t!!))
    (dolist (bit bit-locs)
      (*set word-appears?
            (and!! word-appears?
                   (not!! (zerop!! (load-byte!! *kernel* (!! bit) (!! 1)))))))
      (*if word-appears?
           (*set *score* (+!! *score* (!! word-weight))))))


/* C* code for testing the presence of a single word. */

poly void increment_score_if_all_bits_set
        (mono unsigned word_bit_position[HASH_SIZE], mono int weight) {
    mono j;
    poly bit word_appears = 1;
    for (j = 0; j < HASH_SIZE; j++)
        word_appears &= kernel[word_bit_position[j]];
    if (word_appears)
        score += weight;
}
```

The main search program simply calls this routine once for each keyword in the keyword list.

2.5 Retrieving the Highest Scoring Documents

The code that follows is used to retrieve the *document-id* for each of the highest scoring content kernels in the database. The program returns a list of *document-id*s for the content kernels with the highest scores. The program first retrieves the *document-id* for the highest score, then the next highest score, etc., until a list of length document-count is retrieved. The already-retrieved? flag is set once a processor has had its *document-id* retrieved so it will not be retrieved again.

```
;;; *Lisp code for retrieving documents in order, highest score first.

(*defun retrieve-best-documents
  (let ((top-documents-list nil))
    (*let ((already-retrieved? nil))
      (dotimes (i document-count)
        (*when (not!! already-retrieved?)
          (*when (=!! *score* (*max *score*))
            (*let ((next-highest-document (*min (self-address!!))))
              (setq top-documents-list
                (append top-documents-list
                  (list (pref *document-id* next-highest-document))))
              (setf (pref already-retrieved? next-highest-document) t))))))
    top-documents-list))


/* C* code for retrieving documents in order, highest score first. */

poly void retrieve_best_documents
          (mono document_count, mono unsigned *document_id_array) {
    poly bit already_retrieved = 0;
    mono i;
    for (i = 0; i < document_count; i++) {
        if (!already_retrieved) {
            if (score == (><= score)) {
                processor *next_highest_document = (<>= this);
                document_id_array[i] = next_highest_document->document_id;
                next_highest_document->already_retrieved = 1;
            }
        }
    }
}
```

2.6 Timing and Performance

A production level version of the algorithms described above has been implemented and extensively tested on the Connection Machine system. Performance studies have been done on a database of 15,000 newswire articles, which constitute 40 megabytes of text. An

automatic indexing system, selects the content kernels for each document. The content kernels are about one third of the original size of the text. Surrogate coding compresses the data by another factor of about two. In the system currently in use, the kernels are encoded into as many 1024-bit vectors as are needed at 30 terms per vector. For a long document several vectors are used; additional code, not shown above, is needed to chain the vectors together and combine the results.

Using this encoding, the Connection Machine system is able to retrieve the 20 nearest documents to a 200-word search pattern from a data base of 160 MBytes in about 50 milliseconds. (160 MBytes is equivalent to an entire year of news from a typical newswire.) In this time the Connection Machine system performs approximately 200 million operations for an effective execution speed of 6,000 Mips.

2.7 Summary and Implications

The program is brief because the algorithm is simple. The Connection Machine system is able to match the user's needs directly. It is powerful enough to carry out the algorithm in a straightforward way. The user wants to say to the database "All documents on the same subject as this one, line up in order here." That is exactly the service that the Connection Machine system provides for the user. It broadcasts the contents of the selected document to tens of thousands of processors at once. Each processor decides in parallel how similar its documents are. Then the most similar ones are sorted and presented to the user.

Even larger databases can use the same technique with two enhancements. The first enhancement is the use of a very high-speed paging disk, which allows larger numbers of content kernels to be swapped into the system for searching. The second enhancement is the use of cluster analysis. When the system has many documents on the same subject, it need not store all their content kernels individually. It can store one for the whole cluster, then retrieve the full set of related documents when needed. A single document may, of course, participate in more than one cluster. As the total database size grows, the size of the average cluster grows with it, making this a particularly appropriate technique for large scale databases. The addition of paging and clustering extends the algorithm described above to the 10-gigabyte range and beyond.

Chapter 3

Fluid Dynamics

Fluid flow simulation is a key problem in many technological applications. From the flow of air over an airplane wing to mixing in a combustion chamber, the problem is to predict the performance of a design without building and testing a physical model.

Until recently, fluid flow models were based almost exclusively on partial differential equations, typically the Navier-Stokes equations or approximations to them. These equations are not generally solvable by normal analytical methods. Numerical approximation techniques, such as finite difference methods and finite element methods, have been developed to solve these partial differential equations. All of these methods involve large numbers of floating point operations which require great amounts of fast memory. In addition, obstructions to the flow must usually be mathematically simple shapes.

Recent physics research has suggested that it is possible to make intrinsically discrete models of fluids. The fluids are made up of idealized molecules that move according to very simple rules, much simpler than the Navier-Stokes equations. The models are examples of cellular automata and are particularly well-suited to simulation on the Connection Machine. Cellular automata are systems composed of many cells, each cell having a small number of possible states. The states of all cells are simultaneously updated at each "tick" of a clock according to a simple set of rules that are applied to each cell. This approach involves only simple logical operations and does not require floating point arithmetic. It allows for all obstructions regardless of their shape. In addition, mathematical methods can be used to show that the results of such simulations agree with the results that would be obtained from the Navier-Stokes equations.

3.1 The Method of Discrete Simulation

Discrete simulation is used to model fluid flow on the Connection Machine system. The technique involves six key elements: particles, cells, time steps, states, obstacles, and interaction rules. *Particles* correspond to molecules of a fluid. A particle has a speed and a direction which determine how it moves. A *time step* is a "tick" of a clock that synchronizes the movement of particles. During each time step, particles move one cell in the direction that they are heading. A *cell* is a specific place in the overall region that is being observed. The region is completely filled with cells. Particles can move into and out of each cell during each time step. A *state* is a value assigned to each cell that indicates the number of particles within the cell, and in which directions they are heading. An *obstacle* is a set of special cells that obstruct the natural movement of particles. The *interaction rules* determine the movement of each particle when it shares a cell with one or more other particles. This movement is carried out by updating the state of the cells to reflect the new positions of the particles within the region.

A discrete simulation typically uses fixed cells. The cells never move or change during the simulation. Particles are completely in one cell during a time step, and move completely into the next cell (determined by the interaction rules) during the next time step. During each time step, every cell gathers data about particles heading in its direction from each of its neighboring cells. Based on the interaction rules, each cell determines the direction of its newly acquired particles and updates its own state.

A simulation designer can choose the cell topology and the interaction rules. The cell topology determines how many sides a cell has, and therefore, the directions by which particles may enter and exit. The simulation designer also determines the number of cells in the region being observed, and the average number of particles in each cell. Cellular automata theory provides the background for the simulation designer's decisions. It suggests that a simple cell topology, a huge number of cells and particles, and simple, local interaction rules are the most likely to be successful.

3.2 A Discrete Simulation of Fluid Flow

Thinking Machines is currently simulating fluid flow using a two-dimensional region that is divided into 16,000,000 hexagonal cells. Each cell is assigned to its own Connection Machine processor (using the virtual processor mechanism). The hexagonal mesh is a simple topology that gives the randomness that is required on a microscopic level to get correct results on the macroscopic level.

One of the fundamental reasons for computer simulation of fluid flow is to observe the behavior of a fluid as it flows past an obstacle. In the discrete model, obstacles are groups of cells that particles can not travel through. When a particle approaches an obstacle cell,

it bounces off during the next time step. In order to observe the behavior of a fluid, tens of millions of microscopic particle interactions are simulated. Each individual particle's path through the cells and off of the obstacle cells appears almost random, just as in real fluids. However, when all of the particles' paths are considered, the overall behavior of the model is consistent with the way that real fluids behave. (See references [4,7,14] for further discussion of the use of cellular automata to model fluid flow.)

Individual particles can enter or exit through any of the six sides of each cell. A cell may contain a maximum of one particle heading in each of the six possible directions during a given time step (and so the total number of particles per cell per time step is anywhere from 0 to 6). A particle that has not collided with another particle during a time step will continue moving in the same direction during the next time step. (See figure 3.1.) When particles collide, a simple set of rules determines their new directions, conserving both momentum and the number of particles.

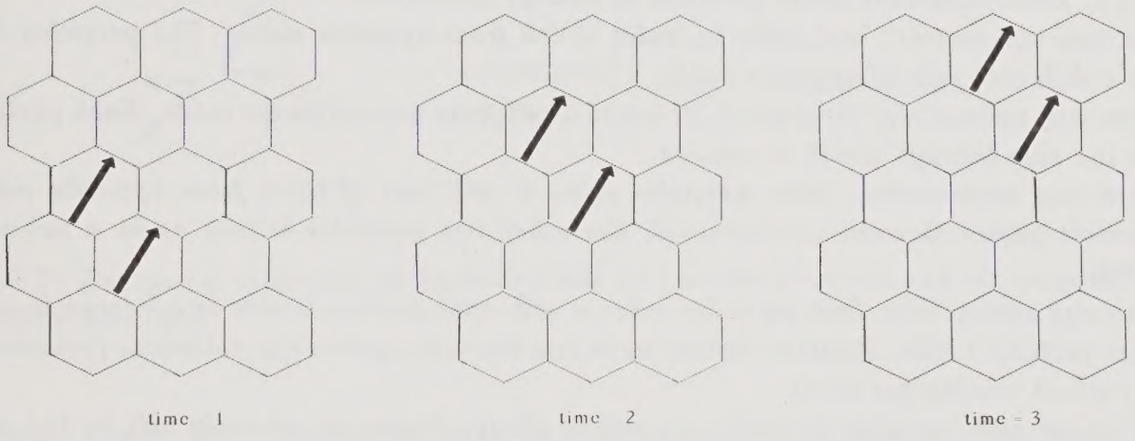


Figure 3.1: *Unless particles are obstructed by an obstacle, or collide into other particles, they continue in the same direction.*

At each time step, every cell updates its state by checking all of its adjoining cells, or neighbors, for particles that are heading in its direction. All cells then update their own states based on the information that they have gathered. In the model currently implemented, there are five situations that cause a particle to change directions: 2-way symmetric collisions, 3-way symmetric collisions, 3-way asymmetric collisions, 4-way symmetric collisions, and collisions with an obstacle cell. (See figure 3.2.)

Although the algorithm is implemented by modeling the individual movements and collisions of tens of millions of particles at each time step, the behavior of the fluid is observed by averaging the behavior of all of the particles in the entire region and by analyzing the

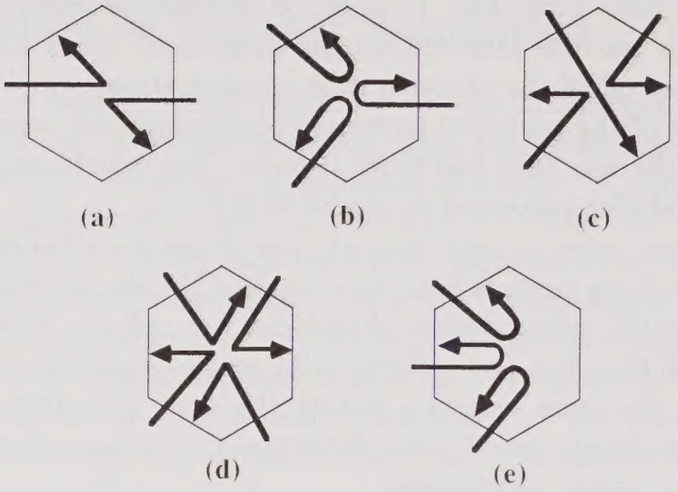


Figure 3.2: *Situations that cause particles to change directions.*
(a) *Two-way symmetric: two particles enter a cell from opposite sides. The particles exit through a different pair of opposite walls.*
(b) *Three-way symmetric: three particles enter a cell from non-adjacent sides. Each particle exits by the side through which it entered.*
(c) *Three-way asymmetric: three particles enter a cell, two of them from opposite sides. One particle passes through unobstructed; the other two particles behave as in a two-way symmetric.*
(d) *Four-way symmetric: four particles enter a cell, each particle's side is adjacent to only one other particle's side. Particles behave as in two two-way symmetric collisions (maximum of one particle exiting per side).*
(e) *Collisions with an obstacle cell: a particle always leaves an obstacle cell by the side through which it entered.*

results over many time steps. In a typical simulation, macroscopic results are gathered by averaging particles together in groups of 20,000. Although each individual particle has only one speed and six possible directions, the average of 20,000 particles provides the full range of possible velocities.

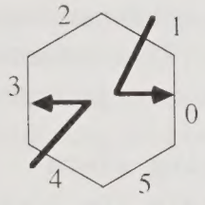
3.3 Implementation on the Connection Machine System

There are two available ways for the Connection Machine system to implement the connections among the hexagonal cells. It can use the full router, setting up six connections for each site, one for each adjacent hexagon. Or it can use its grid, which connects four

adjacent processors directly. The grid network was chosen for this implementation. It is very fast for small data transfers to nearby processors.

Of course, the grid cannot implement hexagonal connections directly. It connects to four adjacent processors, not six. Therefore, two of the six connections require two-step communication (i.e., up one and over one for the diagonal). The simulation program implements this two-step process. Each site can quickly learn the status of its six neighbors and can determine which ones contain particles that are moving in its direction.

Each cell has only 13 bits associated with it: six bits for incoming state (numbered 0–5), six bits for outgoing state (numbered 0–5), and one bit to indicate whether or not it is an obstacle. Each of the six incoming state and six outgoing state bits is dedicated to a particular direction. If a particle is entering or exiting through that direction, then the bit is set to 1, otherwise it is set to 0. (See figure 3.3.)



BITS	0	1	2	3	4	5
INCOMING	0	1	0	0	1	0
OUTGOING	1	0	0	1	0	0

Figure 3.3: *Hexagonal cells with six incoming bits for particle direction and six outgoing bits for particle direction*

```
/* A cell state is represented by a six-bit unsigned integer,
   which can also be regarded as an array of six individual bits. */

typedef union STATE {unsigned:6 Val; unsigned:1 Bit[6];} state;

/* Each processor in the domain "grid" will contain a cell state
   (the outgoing state), another state (the incoming state) used
   for temporary purposes in the calculation, and a bit saying
   whether or not it is an obstacle cell. */

poly state outgoing_state, incoming_state;
poly unsigned:1 obstacle_cell;
```

```
/* The following declares the actual grid of processors. */

processor fluid_grid[ARRAY_X_SIZE][ARRAY_Y_SIZE];

/* Grid is the C pointer type that corresponds to the above array type. */

typedef processor (*grid)[ARRAY_Y_SIZE];
```

At each time step, instructions are broadcast that tell each cell how to gather data about particles heading in its direction. When the cells poll each of their six neighbors for information, they formulate their own 6-bit incoming state. For example, a cell would ask its East neighbor for its outgoing state bit number 3, and would place the answer in its own incoming state bit number 0. It would then ask its NorthEast neighbor for its outgoing state bit number 4 and would place the answer in its own incoming bit number 1. All cells, in parallel, check the state of all six of their neighboring cells. This extreme data level parallelism allows for a large amount of data to be collected in a small amount of time.

```
/* This code is executed within each processor.  Outgoing state
   bits from six neighbors are gathered and placed within the local
   incoming_state array.  Note the use of a C cast expression
   ((grid)this) to create a self-pointer that has a two-dimensional
   array type suitable for double indexing.  (This code actually is
   oversimplified in that it does not handle the boundary conditions
   for cells on the edge of the grid.  Handling these conditions is
   a bit tedious but conceptually straightforward.) */

poly void get_neighbors() {
    incoming_state.Bit[0] = ((grid)this)[ 1][ 0].outgoing_state.Bit[3];
    incoming_state.Bit[1] = ((grid)this)[ 0][ 1].outgoing_state.Bit[4];
    incoming_state.Bit[2] = ((grid)this)[-1][ 1].outgoing_state.Bit[5];
    incoming_state.Bit[3] = ((grid)this)[-1][ 0].outgoing_state.Bit[0];
    incoming_state.Bit[4] = ((grid)this)[ 0][-1].outgoing_state.Bit[1];
    incoming_state.Bit[5] = ((grid)this)[ 1][-1].outgoing_state.Bit[2];
}
```

Once each cell has determined which particles are entering (by collecting its incoming state), it updates its outgoing state to reflect the particle interactions. First, all cells that have their obstacle-bit turned on are instructed to set their outgoing state to be the same as their incoming state (since particles that hit an obstacle bounce back in the same direction).

Next, patterns are broadcast that correspond to each of the possible 6-bit incoming states, followed by the corresponding 6-bit outgoing state. Each cell compares its incoming state to the pattern being broadcast. When there is a match, the cell updates its outgoing state accordingly. For example, a cell with an incoming state of 011011 would then have an outgoing state of 110110 (refer to figure 3.2d).

```
/* The rule table is indexed by a six-bit incoming-state value
   and contains the corresponding outgoing-state values. */

state rule_table[64];

/* Calculate the new outgoing_state for all cells, based on the
   incoming_state and the obstacle_cell bit. */

poly void update_state {
    if (obstacle_cell)
         outgoing_state.Val = incoming_state.Val;
    else outgoing_state.Val = rule_table[incoming_state.Val].Val;
}
```

It is important to note that this trivial, non-computational, table look-up is the driving force of the whole simulation. The Connection Machine system has replaced all of the mathematical complexity of the Navier-Stokes equations with this small set of bit-comparison operations. The simulation is successful because the system can perform this operation on huge numbers of particles in very short amounts of time. It is an example of the Connection Machine system being easier to program because it supports a much simpler algorithm.

3.4 Interactive Interface

A typical "run" of a fluid flow simulation begins by allowing the user to make several choices. The user typically specifies the average number of particles per cell (density) and the average speed and direction of the particles (velocity). Technically this means that the entire region starts out with particles randomly distributed among the cells (based on the density) and moving in a certain overall direction (based on the average velocity). The user also selects or draws one or more obstacles and places them somewhere in the region being observed. All cells that are part of an obstacle have their obstacle bit set. As the simulation runs, new particles are randomly injected from the edges of the region in order to maintain the selected density and velocity. Once the model is running, each cell's state is continually updated, and average results for regions of cells are displayed.

```
/* This is the main computation loop.  At each time step, each
   cell fetches state from neighbors and updates its own state;
   then the results are displayed. */

poly void fluid_flow() {
    for (;;) {
        get_neighbors();
        update_state();
        display_state();
    }
}

/* Execution begins here. */

void start_fluid_flow() {
    /* Initialization. */
    initialize_rule_table();
    initialize_cell();
    /* Activate all processors in fluid_grid
       and then call the function fluid_flow. */
    [[][]fluid_grid].{ fluid_flow(); }
}
```

Figure 3.4: *The formation of a fluid flow phenomenon, called a "vortex street," as fluid flows from left to right past a flat plate.*

3.5 Timing and Performance

A production level version of the algorithm described in this chapter has been implemented and extensively tested on the Connection Machine system. The simulation operates on a 4000×4000 grid of cells, typically containing a total of 32 million particles. The Connection Machine system is able to perform one billion cell updates per second. Figure 3.4 shows several displays from a simulation of 100,000 time steps. Each time step includes approximately 70 logical operations per cell; the simulation therefore required a total of 100 trillion (10^{14}) logical operations. The complete simulation took less than 30 minutes. Current results are very competitive with state-of-the-art direct numerical simulations of the full Navier-Stokes equations.

3.6 Summary and Implications

In addition to providing very accurate simulation of fluid behavior, the Connection Machine method for simulating fluid flow allows scientists to continually interact with the model. Any of the user's original choices may be modified during a run of the simulation, without long delays for new results. Since particles are continually moving through the cells, a new density or average velocity may be established by adjusting the particles being randomly injected from the edges. When a new obstacle is added during a run, the obstacle bits in the appropriate cells are set, and those cells begin to reflect particles. Within less than a minute (a few thousand time steps), results based on the new selections become apparent in the displayed flow.

The algorithm for simulating fluid flow on the Connection Machine system is simple. It overcomes problems formerly associated with computer simulations of fluid flow by using a discrete simulation that takes advantage of the Connection Machine system's inherent data level parallelism. During each time step, every particle can move in the direction it is heading, every cell can evaluate its new particles based on collision rules, and every cell can update its state to reflect the direction of the particles it currently contains. The algorithm involves a small number of instructions executed over a large amount of data. Since the Connection Machine system is able to assign a processor to each data element, and to allow all processors to communicate simultaneously, it has provided the computational power required to provide the ideal solution to this applications need.

Chapter 4

Contour Maps from Stereo Images

Human beings have extremely sophisticated and well-developed visual capabilities, which scientists are just now beginning to understand. Since humans are very good at dealing with visual data, graphics and image processing provide an excellent opportunity for creative partnership between people and computers. An example of this partnership is the widespread use of graphical output for computer applications, such as scientific simulations. The computer does what it does best, computing the results and displaying them in a picture or a movie. Researchers do what they do best, using their sophisticated visual system to make qualitative judgements based on the visual information.

In many important computer applications, however, this partnership breaks down. When the flow of visual data is too large, the human visual system makes mistakes. Often this is simply because humans get tired and lose their concentration when faced with very large and monotonous streams of visual data, not because they are trying to extract information too subtle for current computer science to handle.

4.1 Analyzing Aerial Images by Computer

The analysis of detailed aerial images is an area where increased computer processing is highly desirable. Topographers would like to have the computer partially "digest" the visual data first, presenting only the essential properties of the images to the human user. In some cases, they would like to have the computer go even further, drawing abstract conclusions from raw visual data. Scientific progress in image processing and artificial intelligence has recently made this kind of information processing possible. However, conventional computers cannot keep up with the enormous flow of data that these applications present. Consequently, humans are still doing most of the work in these areas. The partnership has broken down because people are doing what the computer should be doing for them.

Data level parallelism is helping to redress this balance. It is ideally suited to the analysis of multiple images and the detection of subtle differences between them. In particular, it is allowing stereo vision algorithms to be applied to terrain analysis in very high volume applications. Stereo vision is the process by which humans are able to take in two slightly different images (from the two eyes) and use the small differences arising from the two different perspectives to determine the distances to the objects in the field of view. Using the same principle, the Connection Machine system is able to analyze two aerial images to determine the terrain elevation and to draw a contour map. Contrary to the apparent ease with which humans can perform this process, it is a subtle and difficult computational problem which no computer has yet solved perfectly. That is why humans are always involved to "coach" the process. The Connection Machine system, with its natural ability to handle large numbers of images and compare them in great detail, can help to drastically reduce the amount of work people must do in this area.

This chapter describes the underlying algorithms for stereo vision on a data level parallel computer, and shows some of the implementation on the Connection Machine system. Many detailed elements of an actual production system, such as straightening out misaligned images and displaying intermediate results, have been omitted in order to focus on the underlying algorithms. See references [2,3,5,11,12,13] for more information on machine vision and the stereo matching problem.

4.2 Seeing in Stereo

Images are very large, inherently parallel data structures. Therefore the processing of images is an application that is ideally suited for data level parallelism. An image is stored as an array of *picture elements*, or *pixels*. An image with 256 pixels in the vertical dimension and 256 in the horizontal dimension has a total of 65,536 data elements. More detailed images, with 1024 by 1024 pixels, have more than a million data elements. For black and white images, the value stored in each of the pixels is the intensity of light at that point, ranging from pure white through various shades of gray to pure black. (Pixels in color images contain information describing the hue and saturation as well as the brightness.) The contour mapping problem is one of extracting terrain *elevation* information from images that, upon first inspection, contain only information about terrain *brightness* at each pixel.

The term *stereo* means "dealing with three dimensions." *Stereo vision* is "the ability to see in three dimensions." Humans and many animals have the remarkable ability to take in two images, obtained from slightly different perspectives—one from each eye—and fuse them to perceive a three-dimensional world. The difference in perspective causes objects to appear in slightly different places in the two images. The amount of positional difference is related to the distance of the object from the viewer.

Because stereo vision occurs automatically in humans, we tend to be unconscious of the process. A simple demonstration serves as a reminder. Hold a pencil in front of a piece of paper and fix your gaze on the paper. Start to alternately close one eye and then the other, then slowly move the pencil toward your face. Keep the paper stationary and your gaze fixed on the paper while you move the pencil. The paper always seems to shift back and forth by the same small amount, but the closer the pencil moves to you, the more it jumps in position between the two views.

The two images used in a stereo vision system are called a "stereo pair." Figures 4.1 and 4.2 give an example. Figure 4.1 shows a model of some terrain, as seen from an oblique angle. Figure 4.2 shows a stereo pair obtained from directly above the terrain. Figure 4.2 can produce a vivid sensation of depth when observed with an appropriate stereo viewing apparatus.

Figure 4.1: *An oblique view of a terrain model used in a demonstration of the contour mapping algorithm.*

4.3 Finding the Same Object in Both Images

Individual pixels within an image are not reliable indicators of objects. Two pixels, one in each image, can have the same brightness value without being part of the same object. Features larger than individual pixels must be found. The "edges" between areas of different intensities make up an effective set of such features. An edge is a line, usually a crooked line, along the boundary between two areas of the image that have different intensity. Instead of trying to match pixels based on their intensity, the algorithms match them based on the *shape of nearby edges.* The shape of edges is usually much more strongly related to

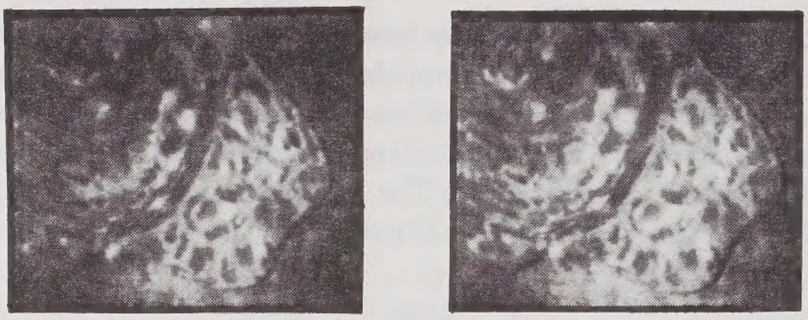


Figure 4.2: *A stereo pair of the terrain in Figure 4.1, obtained from directly above the terrain.*

distinct objects than the simple brightness value.

Figure 4.3 shows an example of edges. These edges were derived from the stereo pair in Figure 4.2.

The process of finding edges falls into the category of image computations called "local neighborhood operations." Individual pixels are classified based on characteristics of a group, or neighborhood, of nearby pixels. Edges are found by having each pixel determine whether the brightness of nearby pixels on one side of it is very different from the brightness of nearby pixels on the other side. This will be the case only for pixels that pass this test: *they must lie between two image regions that are similar within themselves but different from each other.* These *edge pixels* are detected by examining the local neighborhood of every pixel *in parallel*, and storing the ones that pass the test in an array. Typically, only 10 to 20 percent of the pixels in an image get classified as edge pixels.

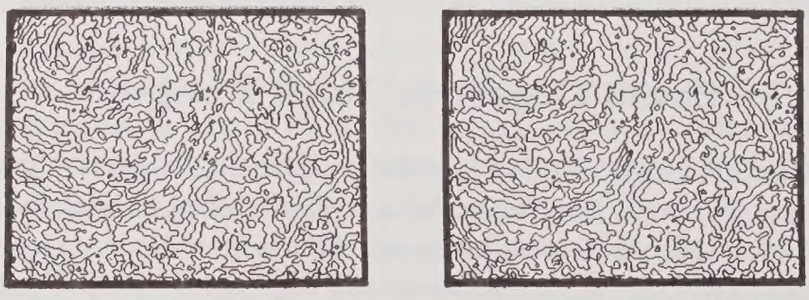


Figure 4.3: *An example of edges. These edges were derived from the stereo pair shown in Figure 4.2. They delineate the boundaries between areas of different intensity.*

4.4 Matching Edges

Even though edges are much more closely tied to objects than simple brightness values, there is still a great deal of work involved in deciding whether an edge in one image corresponds to a particular edge in the other image. Real images suffer from distortions due to several sources. Distortions include random fluctuations or "noise" introduced in the electronic imaging process, relative misalignment between the cameras, and irregular illumination. In addition to these effects, which tend to blur the distinction between edges that match and those that do not, there is a "bad luck" factor: an object or surface marking in one image very often just happens to look like several markings in the other image. For these reasons, the final choice of matches, and therefore the correct positional difference, is always somewhat ambiguous.

If the detection of edges were a perfect process, deciding which positional difference is best for each pixel would be simple. A local neighborhood of edges would align exactly at one relative shift and very little at all the others. Because of the imperfections described above, however, such a high level of precision is impossible. Every neighborhood of edges in one image matches to some extent with many neighborhoods in the other image. The competition is usually very close.

4.5 Measuring Alignment Quality

To resolve the competition, the Connection Machine algorithms hold one of the images stationary and "slide" the other one over it horizontally one pixel at a time. Each time the moving image is slid one more pixel's distance, all the stationary pixels compare themselves to the pixels to which they now correspond in the slid image. They record the presence or absence of an edge alignment in a table in their own memory. Typically, the maximum shift between two images is 30 pixels, so a table of 30 alignment matches is created in the memory of each stationary pixel's processor.

This sliding procedure, using the edges from Figure 4.3, is illustrated in Figure 4.4. Each of the 16 images shows an alignment table entry for each pixel. Black pixels indicate positive alignment table entries, i.e., "match-ups" between the stationary and the sliding images. For example, the 7th image shows alignment-table-slot 7 in each pixel. Thus every black pixel in image 7 corresponds to a match-up between stationary and sliding edges when the relative shift was 7 pixels.

The resulting alignment tables generally show several spurious matches, but also one or two solid ones where the local neighborhood of edges lined up very tightly. When this happens at a pixel, it is a signal that the correct shift (the correct positional difference) for that pixel has been found.

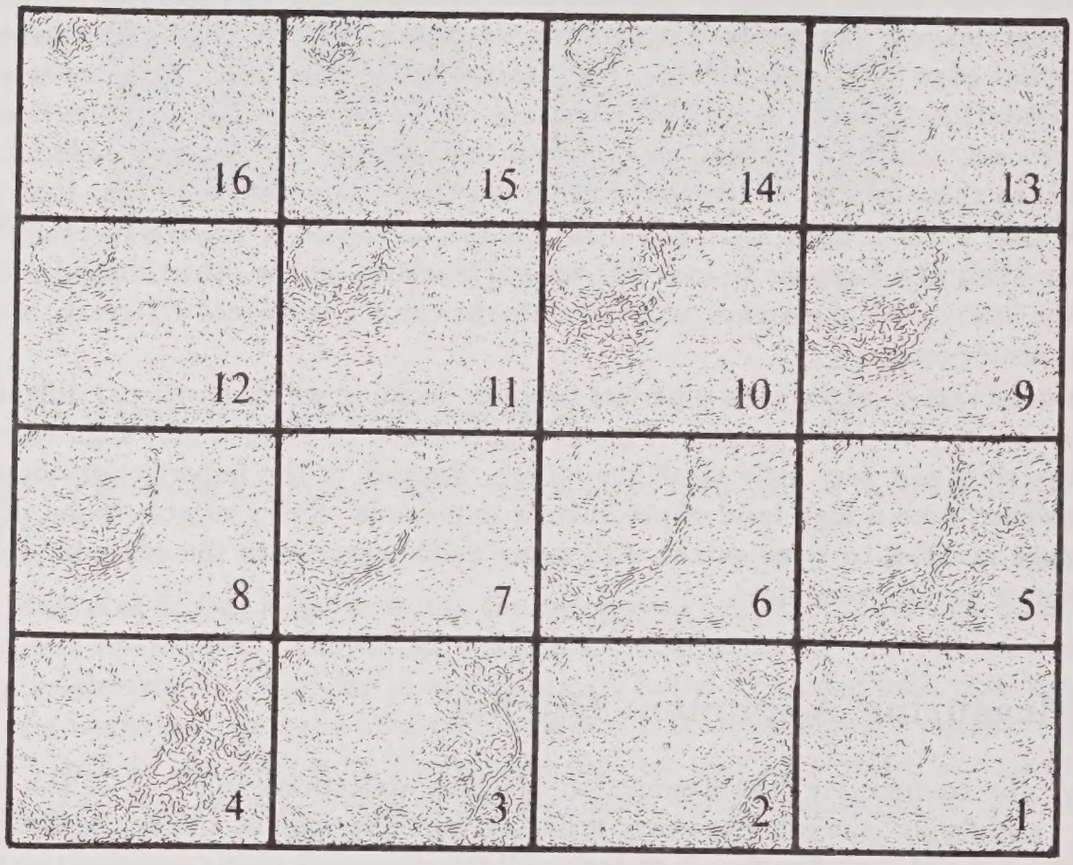


Figure 4.4: *An illustration of the sliding process. Each of these images shows the contents of an alignment-table-slot in each pixel. The Nth image shows slot N in every pixel's alignment table. The dark areas are regions of good alignment, i.e., areas where the same alignment-table-slot is filled in many pixels.*

As in the edge detection process, the alignment quality of every shift position in the alignment table is measured by a local neighborhood operation. In this case, the operation is the following: for each shift position, each pixel processor counts and records the number of matching edge pixels in a small neighborhood around itself. This count or "score" will be high for pixels whose nearby edges are tightly aligned with the edges in the other image *at the same position but displaced by the shift.*

The best shift for a given pixel is determined by comparing the alignment scores at every position in its alignment table. *The shift that has the highest score is chosen as the correct shift for the pixel.* This process takes place in parallel for all pixels; in this way a shift is determined for each pixel.

Areas of tight alignment are clearly visible in Figure 4.4. For example, the small shifts (1 through 4) are tightly aligned over low terrain (refer to Figure 4.1), and the large shifts (13 through 16) are tightly aligned over high terrain. Match-ups in these areas will get high alignment scores because they lie amidst many other match-ups.

4.6 Drawing Contour Maps

The processing described so far yields the shift (or elevation) for every pixel that is part of an edge. These pixels form a "web" of heights that approximates the shape of the terrain, but is not yet smooth and continuous. It is full of holes (where non-edge pixels were) which must be filled in by interpolation.

Interpolation is accomplished by another local neighborhood operation. Each pixel that is not on the web takes on a new elevation which is the average elevation of the pixels in a small neighborhood around it. The neighborhood includes the four pixels above, below, to the left and to the right of the pixel. The pixels that make up the web maintain their original elevations; only the pixels in the holes change their values. This process is repeated or "iterated" a few hundred times.

Pixels that lie in the middle of holes in the web have zero elevation. Therefore, when they become the average of their neighbors, which also have zero elevation, their elevation does not change. However, pixels that lie near the edges of holes in the web have neighbors whose elevation is nonzero. Therefore, when they become the average of their neighbors, they jump to a nonzero elevation. On the next iteration, these new nonzero pixels influence their neighbors, in turn creating new nonzero elevations. Gradually, after a few hundred iterations, the pixels on the web—which remain unchanged throughout the process—"spread" their elevations across the holes in the web, filling it in to create a smooth, continuous surface from which a contour map may be drawn. An example of a contour map is shown in Figure 4.5.

Figure 4.5: *A contour map of the terrain model shown in Figures 4.1 and 2, computed on the Connection Machine system.*

4.7 Finding Edges on the Connection Machine System

A pixel is classified as an edge pixel if it lies between two image regions that are similar within themselves but different from each other. This is the program that performs the edge classification operation.

```
(*defun find-edges-between-left-and-right!! (brightness-pvar threshold)
  (*let* ((average-brightness-on-the-left
           (/!! (+!! (pref-grid-relative!! brightness-pvar (!! -1) (!! -1))
                     (pref-grid-relative!! brightness-pvar (!! -1) (!! 0))
                     (pref-grid-relative!! brightness-pvar (!! -1) (!! 1)))
                (!! 3.0)))
          (average-brightness-on-the-right
           (/!! (+!! (pref-grid-relative!! brightness-pvar (!! 1) (!! -1))
                     (pref-grid-relative!! brightness-pvar (!! 1) (!! 0))
                     (pref-grid-relative!! brightness-pvar (!! 1) (!! 1)))
                (!! 3.0)))
          (average-brightness-overall
           (/!! (+!! average-brightness-on-the-left
                     average-brightness-on-the-right)
```

```
                (!! 2.0))))
    (if!! (>!! (absolute-value!! (-!! average-brightness-on-the-left
                                       average-brightness-on-the-right))
               (*!! (!! threshold) average-brightness--overall))
          (!! 1)
          (!! 0))))
```

The preceding program sequence calculates the average brightness in a small region to the left (i.e., with relative x-coordinate -1, and relative y-coordinates -1, 0, and 1) and the average brightness in a small region on the right side (with relative x-coordinate 1) of each pixel. If, at any particular pixel, the difference between these averages is greater than the specified threshold, then the pixel is marked with a one, meaning that it is an edge pixel. Otherwise it is marked with a 0. The threshold is multiplied by the overall average brightness, a process called "normalization." With normalization, the threshold adapts to the image, becoming small in regions where the image is generally dark, and large where the image is generally bright.

Since this program compares regions on the left and right sides of a pixel, it works only for edges that are more or less vertical. It is easy to write a program that finds horizontal edges by having it compare small regions on the top and bottom of a pixel, in the same way that this program compares regions on the left and right. The same could be done edges in both diagonal directions. The four programs may then be combined to find *all* edges in the following way:

```
(*defun find-all-edges!! (brightness-pvar threshold)
  (if!! (or!! (=! (!! 1) (find-edges-between-left-and-right!!
                             brightness-pvar threshold))
              (=! (!! 1) (find-edges-between-above-and-below!!
                             brightness-pvar threshold))
              (=! (!! 1) (find-edges-between-upper-left-and-lower-right!!
                             brightness-pvar threshold))
              (=! (!! 1) (find-edges-between-lower-left-and-upper-right!!
                             brightness-pvar threshold)))
        (!! 1)
        (!! 0)))
```

4.8 Matching Edges on the Connection Machine System

The following program sequence implements the sliding procedure described above. One of the edge images is held stationary and the other edge image is moved across it horizontally,

one pixel at a time. At each relative shift (1, 2, ..., 30), each processor records whether an edge match has been found in the sliding image. This information is stored in a pvar that represents one of the alignment tables discussed above. All of the alignment tables are stored in the Connection Machine memory at the same time.

```
(defvar *array-of-pvars-holding-matches-at-each-shift* (make-array 30))
  ;;; This is just a regular Lisp array, but each element of this
  ;;; array will be a pvar.  Notice that we'll try to find positional
  ;;; differences of up to 30 pixels.  (Note: each one of the pvars
  ;;; in this array will hold an "alignment-table-slot" for every pixel,
  ;;; as discussed in the text).

(*defun fillup-pvars-wherever-edges-align (left-edges right-edges)
  ;; This program records the edge-pixel match-ups at every shift;
  ;; that is, this program creates "match-up images," as shown in
  ;; Figure 4.4.
  (dotimes (i 30)
    (aset (if!! (=!! left-edges
                     (pref-grid-relative!! right-edges (!! i) (!! 0))
                     )      ; ^This PREF-GRID-RELATIVE!! accomplishes
                (!! 1)      ; the "sliding" process.
                (!! 0))
          *array-of-pvars-holding-matches-at-each-shift*
          i)))
```

The next step in the process is to decide at each pixel position which shift produced the best match-up. Most locations will contain a somewhat random pattern of match-up pixels. However, at some locations, the local neighborhood of match-ups will be very dense and regular, indicating that the shift responsible for that match-up image is probably the correct shift for that neighborhood.

The following *Lisp program measures the density or alignment quality of every neighborhood. It does so by counting the number of 1's (match-ups) in a square around each pixel. The counting process is accomplished in parallel, for all pixels at once, on the Connection Machine system.

```
;;; The square for each pixel is to be centered on that pixel.
;;; Because a DOTIMES loop always produces values starting at zero,
;;; it is necessary to subtract one-half the width of the square
;;; from the loop variable in order to get relative indexes that
```

```
;;; are centered on zero.

(*defun add-up-all-pixels-in-a-square (pvar width-of-square)
  (let ((one-half-the-square-width (/ width-of-square 2)))
    (*let ((total (!! 0)))
      (dotimes (relative-x width-of-square)
        (dotimes (relative-y width-of-square)
          (*set total
                (+!! total
                     (pref-grid-relative!!
                        pvar
                        (- relative-x one-half-the-square-width)
                        (- relative-y one-half-the-square-width))))))
      total)))
```

At this point, it is a simple matter to record the alignment quality or score for every pixel.

```
(defvar *array-of-pvars-holding-scores-at-each-shift* (make-array 30))
  ;;; Another Lisp array holding *Lisp pvars.
```

The next step is to fill all the elements of the Lisp array with *Lisp pvars. The Nth element of the Lisp array holds a pvar containing the scores, or alignment qualities, of all the matches that occurred when the edge images were shifted by N pixels relative to each other. (Note that this program records scores only at locations where match-ups occurred. Other locations have no score, which reflects our original intention of matching *edges*, not the holes between them.)

```
(*defun fillup-pvars-with-match-scores (width-of-square)
  ;; WIDTH-OF-SQUARE will typically be 21.
  (dotimes (i 30)
    (*let ((sum-of-all-nearby-pixels
             (add-up-all-pixels-in-a-square
               (aref *array-of-pvars-holding-matches-at-each-shift* i)
               width-of-square)))
      (*if (=!! (aref *array-of-pvars-holding-matches-at-each-shift* i)
                (!! 1)) ;;; Record a score wherever there was a match-up.
           (*set sum-of-all-nearby-pixels
                 *array-of-pvars-holding-scores-at-each-shift*
                 i)))))
```

Now that the score for every match-up has been recorded, there is only one more step required to establish which of the match-ups is correct. The following *Lisp program loops through all the shifts, keeping track of the best score at each pixel. The shift that produced the best score at each pixel is recorded as the "winning shift."

```
;;; This function computes the web of known shifts.  Recall that
;;; the shift at each pixel corresponds directly to the elevation.

(*defun find-the-shifts-of-the-highest-scoring-matches ()
  (*let ((best-scores (!! 0))
         (winning-shifts (!! 0)))
    ;; The following DOTIMES loop makes sure that each
    ;; pixel in the BEST-SCORES pvar contains the maximum
    ;; score found at any shift.
    (dotimes (i 30)
      (*if (>!! (aref *array-of-pvars-holding-scores-at-each-shift* i)
                best-scores)
           (*set best-scores
                 (aref *array-of-pvars-holding-scores-at-each-shift* i))))
    ;; The following DOTIMES loop records a "winning"
    ;; shift at every pixel whose score is the best.
    (dotimes (i 30)
      (*if (=!! (aref *array-of-pvars-holding-scores-at-each-shift* i)
                best-scores)
           (*set winning-shifts (!! (1+ i)))))
    winning-shifts))
```

4.9 Drawing Contours on the Connection Machine System

A contour map cannot be constructed without a smooth, continuous surface on which to draw the lines. All of the processing so far has produces a web of known elevations (returned by the last *Lisp function above). Interpolation across the holes in the web produces a continuous surface.

```
(*defun fill-in-web-holes (web-of-known-elevations times-to-repeat)
  ;; Each time through the loop, every pixel not on the web (i.e.,
  ;; every pixel that is not zero to begin with) takes on the
  ;; average elevation of its four neighbors.  Therefore, the web
  ;; pixels gradually "spread" their elevations across the holes,
  ;; while they themselves remain unchanged.
  (dotimes (i times-to-repeat)
    (*let ((not-fixed (zerop!! web-of-known-elevations)))
      (*if not-fixed
         (*set web-of-known-elevations
               (/! (+!! (pref-grid-relative!!
                         web-of-known-elevations
                         (!! 1) (!! 0))          ;Neighbor to the right
                        (pref-grid-relative!!
                         web-of-known-elevations
                         (!! 0) (!! 1))          ;Neighbor above
                        (pref-grid-relative!!
                         web-of-known-elevations
                         (!! -1) (!! 0))         ;Neighbor to the left
                        (pref-grid-relative!!
                         web-of-known-elevations
                         (!! 0) (!! -1)))        ;Neighbor below
                   (!! 4))))))
   web-of-known-elevations) ;;; this is now a more or less smooth surface.
```

The following code takes the smoothed-out web and constructs a contour map in the form of a plane of black-and-white pixels suitable for display on a graphics device.

```
(*defun draw-contour-map (number-of-contour-lines
                          pvar-of-smooth-continuous-elevations)
  ;; The idea is to divide the whole range of elevations into
  ;; a number of intervals, then to draw a contour line at every
  ;; interval.
  (let* ((max-elevation (*max pvar-of-smooth-continuous-elevations))
         (min-elevation (*min pvar-of-smooth-continuous-elevations))
         (range-of-elevations (- max-elevation min-elevation))
         (contour-line-interval (/ range-of-elevations
                                   number-of-contour-lines)))
```

```
;; Now the variable CONTOUR-LINE-INTERVAL tells us how many
;; elevations, or shifts, to skip between contour lines.
(if!! (zerop!!
       (mod!! (-!! pvar-of-smooth-continuous-elevations
                   (!! min-elevation))
              (!! contour-line-interval)))
      (!! 1)    ;; This IF!! draws all the elevation contours
      (!! 0)))) ;; at once, returning a bit map suitable for
                ;; for immediate display.
```

4.10 Timing and Performance

A production level version of the contour mapping algorithm described in this chapter has been implemented and extensively tested on the Connection Machine system. Parameters such as the size of the images and the range of positional differences ("shifts") are variable, depending on the application. A typical program run processes images containing 512×512 (262,144) pixels, while allowing for positional differences from 0 to 30 pixels. In such a mode, the Connection Machine system performs approximately two billion (2×10^9) operations during the most time-consuming phase of the algorithm, the so-called "inner loop," in which the match-ups are detected and their alignment quality is measured. This inner loop is executed in less than two seconds.

4.11 Summary and Implications

Contour mapping using stereo vision is an example of an image processing application that is sophisticated and computationally expensive. The Connection Machine system, because it readily accommodates itself to the inherently parallel structure of image data, made it easy to conceptualize and to program the contour mapping algorithm. The simplicity and brevity of the programs shown above is evidence of this natural fit.

The raw speed of the Connection Machine system is as valuable as its architecture. The system can extract elevation information from large amounts of visual data at very high rates. This speed allows scientists and engineers who are developing new techniques in computer vision to try their ideas "on the fly." A short turnaround time for experimenting with new ideas is essential for the rapid development of the field of computer vision. The effects of various program modifications are realized almost instantaneously. The system's computational power is a valuable aid in the design and implementation of sophisticated algorithms.

Chapter 5

The C* Programming Language

C* (pronounced *see star*) is a simple extension to the C programming language [6,10] that exploits the power of the Connection Machine architecture. C* is (almost) a strict extension of C; any valid C program, if it avoids the use of a small number of C* reserved words, is also a valid C* program. A few new features of the language serve to indicate where data is stored and which operations are executed in parallel in the Connection Machine network.

5.1 C* Extensions

In order to indicate whether a variable is located on the host or in the Connection Machine memory, two storage class identifiers `mono` and `poly` have been included in C*.

```
mono int x;     /* x resides in the host memory */
poly int y;     /* y resides in the Connection Machine memory */
```

The modifier `poly` declares variables present in all processors.

The majority of parallel code is standard C code. Parallel functions are simply distinguished by the identifier `poly`. It is a mark of the general-purpose nature of the Connection Machine architecture that the full C language is available for programming the processors of the Connection Machine system. Likewise, it is a mark of the simplicity of the architecture that the C language suffices for this task. In fact, no new language features need to be introduced in order to perform parallel control flow, interprocessor communication, and memory allocation. The real power of C* comes from the natural parallelization of familiar constructs of C.

5.1.1 Parallel Control Flow

Inside of a parallel function, the normal C control-flow statements, such as `if` and `while`, work as expected. This is perhaps unexpected to someone experienced with other parallel languages. For example, an `if` statement may have a conditional expression whose value is different in different processors:

```
poly salary;
...
if (salary <= 0)
    salary = fixup_salary();
```

It would clearly be an error for *all* processors to make the call to `fixup_salary`. The way C* handles such a statement is to reduce the active set of processors, by temporarily inactivating all those whose `salary` variables are positive. The body of the `if` statement is run, and then the original active set is restored. Such conditional statements can be nested to any degree.

The `while` statement can also operate in parallel. At each evaluation of the loop's conditional expression, more processors can drop out of the active set; they stay inactive until the loop is finished. Finally, when all processors are finished with the loop, the statement is done, and the original active set is restored. For example:

```
poly resumes_to_read;
...
while (resumes_to_read > 0) {
    /* Read ten resumes at a time. */
    resumes_to_read -= 10;
    ...
}
```

In this case, all processors with `resumes_to_read` between 1 and 10 execute the loop body exactly once.

All other standard C control constructs are handled in similar ways in C*; even *goto* is accommodated. The program behaves as if the standard C code were running separately in each processor, with processors that are doing the same thing doing it at the same time.

5.1.2 The Selection Statement

In order to execute code in a selected set of processors, an additional statement called the *selection statement* is included in C*. Selection statements may be used within any C* function. The selection statement has the form:

```
[selector].statement
```

The selector indicates a set of processors. These are activated, and the `statement` is executed within those processors. For example, given the following declaration,

```
processor managers[100];
```

the following statement

```
[[100]managers].{ salary *= 1.06; }
```

or, more simply,

```
[[]managers].{ salary *= 1.06; }
```

selects all 100 of the `managers`, and gives them a six percent raise. The code:

```
[[50]managers].{ salary *= 1.11; }
```

gives the first 50 an eleven percent raise, while this:

```
[managers[0],managers[2]].{ salary -= 1000; }
```

singles out the first and third managers for a pay cut. (More complicated forms of selection are also available.)

5.1.3 Computation of Parallel Expressions

C* extends the meaning of C expressions to parallel computations by means of two simple rules. The first rule says that if a single value (typically of storage class `mono`) is combined with a parallel value (of class `poly`), the single value is first replicated to produce a `poly` value. (In hardware terms, the single value is *broadcast* to all relevant processors.) For example, in the expression `(salary > 20000)`, the single value 20000 is replicated to match the parallel variable `salary`. This rule is an addition to the rules of "usual conversions" in plain C.

The second rule says that an operation on a parallel value (or values) must be processed *as if* only a single operation were executed at a time, in some serial order. In the expression `(salary > 20000)` it is *as if* we took first one `salary` value and compared it to 20000, then another, and so on, doing the comparisons one at a time.

Fortunately, we can analyze the > operation and determine that doing all the comparisons at once will produce the same result, because doing so will not affect the outcome. This is hardly surprising, and it is exactly the effect we want anyway, so why do we have the "as if serial" rule at all? It is because some operators *do* have side effects: assignment operators. Consider the expression

```
total_payroll += salary;
```

Now `total_payroll` is a single value (what in C is called an *lvalue*, because it occurs on the left side of an assignment). By the first rule it is replicated. We then have many assignments to perform, one for each value in the parallel value `salary`:

```
total_payroll += salary_1;
total_payroll += salary_2;
total_payroll += salary_3;
        .
        .
        .
```

The second rule guarantees that the program behaves *as if* all of these assignments were performed in some serial order. *Which* order does not matter; the result is the same. The point is that if these assignments were executed in parallel some updates might be lost; but C* guarantees that *all* the `salary` values will be correctly added into `total_payroll`. (Doing this efficiently is handled by the C* implementor.)

A C assignment operator may be used as a unary operator in C* to reduce a parallel value to a single result that may be further operated upon. For example,

```
(+= salary)
```

adds up the salaries for all persons for which processors are active, and

```
(+= salary)/(+= ((poly) 1)))
```

computes the average of all salaries because the expression

```
((poly) 1)
```

makes a 1 for every active processor and

```
(+= ((poly) 1)))
```

adds up all the 1's, thereby counting all the active processors.

In C*, "<>" is the "minimum" operator and "><" is the "maximum" operator. The expression "`a >< b`" means the same as "`(a > b) ? a : b`". The assignment operators `<>=` and `><=` are also defined: "`a <>= b`" assigns `b` to `a` if `b` is less than `a`. The expression `(><= salary)` finds the largest salary, and `(<>= salary)` finds the smallest salary.

5.1.4 Data Movement

C* has no language extensions to handle data movement or interprocessor communication *per se*. Instead, the normal C operations are used; the Connection Machine architecture allows random access to the desired datum, wherever it is in the system.

Within the code of a `poly` function, the keyword `this` is a C* reserved word whose value is a pointer to the currently executing processor. This value is sometimes called the *self-pointer*. If many processors are executing, each will have its own self-pointer. References to the processor's variables implicitly refer to the self-pointer: saying `salary` is the same as saying `this->salary`. Explicit references to `this` are useful for accessing the memory of neighboring processors through indexing.

The key point is that any processor may contain a pointer to data in the memory of any other processor, and access through that pointer is supported by the Connection Machine router. All interprocessor communication can therefore be expressed in C* merely by the usual explicit and implicit pointer indirection mechanisms. For example, to increment a neighbor's `salary` field, and then decrement one's own based on the result, the following code might be used:

```
this[1].salary += 1000;
salary -= this[1].salary * .10;
```

Similar expressions can also be used to broadcast data throughout the system, to transfer data between the host and Connection Machine processing network, or to collect data from many sources into one location.

5.2 Summary

The C* language is a version of the standard C language suitable for programming the Connection Machine system. Because of the simplicity and power of the Connection Machine architecture, C* itself is a simple yet powerful extension of C. The Connection Machine memory is treated as a large section of host-accessible memory with active objects stored in it. Because standard C is already excellent at manipulating structures, pointers, and the like, relatively few new language features are needed to deal with the Connection Machine architecture. All the familiar C language constructs acquire the power of parallelism easily and naturally.

Chapter 6

The *Lisp Programming Language

*Lisp (pronounced *star lisp*), is an extension of Common Lisp [9], a standard dialect of Lisp that is found on a variety of computer systems. Lisp has many features that are common to most programming languages, but its unusual structure and syntax make the programs a bit difficult to read for someone who has mainly had experience with block structured languages such as FORTRAN or C.

This chapter covers both Lisp and *Lisp in sufficient depth to make it possible to understand the program examples in this book. See references [9,15,16] for a deeper understanding of the Lisp language and its structure.

6.1 Fundamentals of Lisp

What most people remember about Lisp is that it uses lots of parentheses. And it is true—Lisp does. But it is not necessary to understand the full implications of the parentheses to understand the sample programs. Roughly, in a Lisp expression the first thing that comes after the open parenthesis is the function name, and after that are the arguments. So `(+ 7 A)` would call the function `+`, which adds 7 and the value of the variable `A`, and returns the result.

Lisp function calls can be nested as they can in other languages. For example:

```
(* 5 (+ 1 2 3))
```

would first add together 1, 2, and 3, and then multiply the result by 5, giving 30.

Most Lisp programs are indented to help reveal their structure and to show how many levels deep parentheses have been nested. Expert Lisp programmers keep their code properly indented, and rely on the indentation as much as the parentheses when reading code.

6.1.1 Lisp Functions

Functions are the program building blocks of Lisp. Unlike many other programming languages, Lisp does not have a main program followed by a series of functions. In Lisp everything is a function, and programs are executed by invoking those functions from an interactive Lisp interpreter.

The Lisp function-defining operation is called `DEFUN`. The first argument to `DEFUN` is the name of the function that is being defined, the second a list of its arguments; these are followed by the operations to be performed. For example:

```
(defun add-three (x) (+ x 3))
```

defines a function named `add-three` that takes one argument named x, and the operation that is performed by the function is `(+ x 3)`.

6.1.2 Variables

It is not necessary in Lisp to predefine variables, but it is often done for clarity. The mechanism is straightforward:

```
(defvar a 25)
```

defines a variable named a with an initial value of 25. Variables defined with `defvar` are global variables that can be accessed by any function at any time.

Temporary variables are defined in Lisp with the `let` operation, which takes a list of variable-value pairs, and is followed by a sequence of operations to be performed. For example,

```
(let ((temporary 25)
      (x 49))
  (print (+ temporary x))
  (print (* temporary x)))
```

allocates two temporary variables `temporary` and x, assigns them the values 25 and 49 respectively, prints their sum and product, and then deallocates them when the `let` is exited.

Variables have their value set with the `setq` function which takes as its arguments a variable name and a value. So

```
(setq b 34.5)
```

sets the variable `b` to 34.5.

6.1.3 Program Control Structure

The `if` construct is a simple method for conditionally controlling the flow of a program; it is used in several places in the example programs. It takes a test clause, an expression to evaluate if the result of evaluating the test clause is true, and, optionally, an expression to evaluate if the result is false. The following simple example shows how `if` is used.

```
(if (= a 10)
  (print "a is 10")
  (print "a is not 10"))
```

Several of the examples use `dotimes`, a facility for executing a series of expressions a specified number of times. As an example,

```
(dotimes (j 10)
  (print j))
```

prints the integers from 0 to 9.

6.2 *Lisp Extensions

A *Lisp program looks much like an ordinary Lisp program. The biggest distinction is that *Lisp operations manipulate data stored in the Connection Machine hardware, while Lisp operates exclusively on the host processor. There are no instructions stored in the Connection Machine processors; instructions are generated from the *Lisp program and broadcast to the Connection Machine system.

The names of most *Lisp functions either begin with an "*" or end in "!!" (meant to look like two parallel lines, and pronounced *bang bang*) which means that they perform operations on parallel variables. This is only a naming convention and does nothing but distinguish functions that work with the Connection Machine system and parallel variables from functions that don't. User programs may also follow the convention, but it is not a requirement.

This section describes enough *Lisp to make the example programs understandable. As part of that, it is first necessary to describe a few of the fundamental features of the Connection Machine system.

6.2.1 Processors

A *processor* is the entity that operates on data in parallel. Each processor has a unique address that allows it to be directly accessed. The address is made up of one or more numbers depending how many dimensions the Connection Machine hardware is simulating. A

one dimensional machine would take one number as an address, a two dimensional machine two numbers, etc. *Lisp has instructions that can directly access data in the Connection Machine processors via these addresses.

6.2.2 Parallel Variables

The parallel variable mechanism is one of the key programming differences between *Lisp and sequential programming languages. A thorough understanding of what parallel variables are and how they work is crucial to understanding the example *Lisp programs in this document.

On a serial machine a variable may have only one value at a time. On the Connection Machine system a parallel *Lisp variable has as many values as there are processors. Descriptors for parallel variables, or *pvars*, reside on the host computer, and the values of those parallel variables are in the Connection Machine memory.

The *Lisp expression for defining a pvar is similar to the Lisp mechanism for allocating a variable. The expression

```
(*defvar b (!! 5))
```

defines a pvar named `b` which has a value of 5 on every processor in the machine. The function `*defvar` is the parallel version of Lisp's `defvar`. The expression

```
(!! 5)
```

is the part of the `defvar` that actually does the allocation of a field with a value of 5 in every Connection Machine processor.

Values are retrieved from processors with the `pref` function. For example,

```
(pref b 7)
```

would return the value of pvar `b` in processor 7. Setting a value in a processor is accomplished with the Lisp `setf` function.

```
(setf (pref b 3) 10)
```

would set the value of pvar b to 10 in processor 3. The first argument to `setf` describes how to access the field that is going to be altered and the second argument is the new value of the field.

The following series of *Lisp expressions show in some detail how to allocate and use pvars.

First define some pvars:

```
(*defvar a)
```

```
(*defvar b (!! 5) "This is a documentation string.")
(*defvar c (!! -2.67))
(*defvar d t!!)
(*defvar e (1+!! (self-address!!)))
```

These statements created five pvars. The last four have been initialized with specific values: `b` is a Lisp symbol that has as a value a pvar whose contents is the integer 5 in each processor, `c` contains the floating point number -2.67 in each processor, `d` contains the boolean value true in each processor, and `e` contains the address of the next higher processor. The function `self-address` is a function that returns a pvar which contains the address of the selected processor.

Now read some of the values using `pref`.

```
(pref c 0)
```

returns the lisp value -2.67 since that is what is contained in pvar `c` in processor 0.

```
(pref d 365)
```

returns the lisp value `t` since that is what is contained in pvar `d` in processor 0.

Now do some arithmetic on these pvars:

```
(*set a (+!! b c))
```

will set the contents of pvar `a` to be the sum of the contents of pvar `b` and pvar `c`. Notice that `c` contains floating-point values. The integers contained in `b` are converted to floating-point numbers and the result in `a` will be floating point as well. Expressions can be nested:

```
(*set a (-!! b (*!! a (!! 2))))
```

This expression sets `a` to the difference of `b` and twice `a`. This simple expression could cause thousands of such operations to go on simultaneously! The expression `(!! 2)` returns a pvar that is 2 in all processors.

This point is important. The expression

```
(+!! a 2)
```

is an incorrect *Lisp expression. The variable `a` is a pvar, whose values are stored on the Connection Machine system, while the integer 2 is a Lisp object stored on the front end system. It is necessary to convert the 2 to a parallel value before doing any parallel computation.

6.2.3 Accessing Pvars Relative to a Grid

Two of the example programs, fluid flow and stereo matching, make heavy use of the Connection Machine system's grid mechanism, which facilitates communications between processors for problems with two-dimensional data structures. For example say `image` was a pvar containing a two-dimensional image. The following expression would shift the entire image over by one pixel in the x direction and place the result in `shifted-image`:

```
(*set shifted-image (pref-grid-relative!! image (!! 1) (!! 0)))
```

in this example the `(!! 1)` specifies that there is a shift of 1 in the x-dimension, and the `(!! 0)` specifies that there is no shift in the y-dimension.

6.2.4 Selection

In *Lisp it is possible to do an operation in a selected subset of all processors. The *Lisp function `*when` is used to do that selection. For example:

```
(*when (=!! a (!! 5))
  (*set a (+!! (!! 2))))
```

adds two to a in all processors in which a has a value of 5.

6.2.5 *Lisp Programs

*Lisp programs are defined in much the same way that Lisp functions are defined. The main difference is that `*defun` is used instead of `defun` to define functions that either take a parallel variable as an argument or return a parallel variable as a result.

6.3 Summary

*Lisp is a simple extension to Common Lisp that integrates the Connection Machine system into an ordinary serial programming environment. For someone familiar with Lisp, the essentials of *Lisp can be learned and put to productive use within a few hours.

Chapter 7

The Connection Machine System

The Connection Machine system from Thinking Machines Corporation is the first computer to implement data level parallelism in a general purpose way. It combines a very large number of processors with the communications capability necessary to match data topologies exactly. This chapter describes the hardware implementation of the Connection Machine system.

7.1 Connection Machine Internal Structure

As described in Chapter 1, the Connection Machine system operates by receiving a stream of instructions from its front end computer. A microcontroller receives the instructions, expands each of them into a series of machine instructions, then broadcasts the machine instructions, one at a time, to all processors at once. The instructions coming in from the front end are referred to as "macro-instructions." The instructions broadcast to the individual processors are called "nano-instructions." Macro-instructions are similar to assembly language instructions on a conventional machine. They are the instruction codes produced by the Connection Machine language processors. In the sections that follow, names of macro-instructions appear in italics.

The Connection Machine system includes 65,536 physical processors, but may be configured for a much larger number of logical processors by means of the *cold-boot* command. *Cold-boot* takes two arguments that allow a two-dimensional array of virtual processors per physical processor. *Cold-boot*$(4,4)$, for example, sets up the machine in the million-processor mode (or, more precisely, the 1,048,576 processor mode) because each of the 65,536 processors will simulate 16 (4×4) virtual processors. The same number of virtual processors could be established by the command *cold-boot* $(16,1)$. Since virtual processors are so commonly used, they are referred to simply as "processors". Where it is necessary to refer to one of

the 65,536 hardware processors, the term "physical processor" is used.

Each physical processor has 4096 bits of memory, totalling 32 megabytes for the machine as a whole. In the million-processor mode, each processor has 256 bits of memory. Memory is divided into a data area and a stack area, with the layout being the same in each processor. A single, system-wide register, the stack limit, defines the boundary between stack space and data space. The stack pointer is also a system-wide register. The stacks in all processors act in unison.

Memory is bit-addressable; all data fields are of arbitrary length. For numeric computing there are three standard formats: unsigned-integer, signed-integer, and floating-point. Each is of arbitrary length. In particular, floating-point numbers can be of any length. Picture and word data are of arbitrary format and length.

A complete Connection Machine memory address has three parts. The first part indicates a physical processor. The second part indicates one of the virtual processors simulated by that physical processor. (This part is empty if there is only one virtual processor per physical processor.) The third part is an address within the memory of that virtual processor.

Data may be exchanged between the Connection Machine memory and the front end in any of three ways: slicewise, processorwise, and arraywise. *Read-slice* reads a single bit of information from the memory of each of a series of consecutive processors, assembles them into a signed integer, and passes the integer to the front end. *Write-slice* moves data from the front end to the Connection Machine memory. Slice operations are typically done 16 or 32 processors at a time. *Read-processor* and *write-processor* move a single field between the front end and a single processor. *Read-array* and *write-array* move arrays of fields between the front end and a set of contiguous processors.

7.2 Connection Machine Instruction Flow

All instructions flow into the Connection Machine hardware from the front end. These macro-instructions are sent to a microcontroller, which expands them into a series of nano-instructions. Some expand into just a few nano-instructions. Others expand into hundreds or thousands. It is also possible to feed nano-level instructions to the microcontroller and control the hardware directly. It is not, however, efficient to do so, because the front-end cannot supply these instructions rapidly enough to keep the system busy. (Direct control of the hardware from the front end is provided primarily so that the front end can support debugging and diagnostic aids.)

Nano-instructions are broadcast to all processors in parallel. Processors, however, have the option of "sitting out" a series of instructions. A one-bit flag within each processor, the *context flag*, determines whether that individual processor will respond to the instruction

or not. Most of the instructions discussed in this chapter are "conditional" in the sense that they take effect only in the processors that are *active*, that is, whose *context flag* is 1.

The Connection Machine system is implemented with four physical microcontrollers, one for each section of 16,384 processors. If the system has a single front end, that front end is connected to all four microcontrollers and therefore drives all 65,536 processors. A system may be configured with up to four front ends. A crossbar switch called the Nexus makes the connections between front ends and microcontrollers. It is possible, therefore, to have four users operating simultaneously. Each works at a separate front end, and each has a separate instruction stream executing in a section of the system's processors. The examples in this chapter, however, assume that the system is operating with a single front end.

7.3 Computational and Global Instructions

Computational instructions operate on signed integers, unsigned integers, and floating-point values. They include unary operators such as *not*, *negate*, *absolute value*, and *square root*. All standard binary operators such as *add*, *subtract*, *multiply*, *divide*, *compare*, and *shift* are included. These instructions operate in all processors simultaneously; each processor uses the data that is stored in that processor's memory.

The *random* instruction places an independently chosen pseudo-random number in each processor. Two processors may or may not be assigned the same random value.

Global instructions produce a single result from data items stored in the memories of all selected processors. *Global-logior*, for example, takes the inclusive OR of a field in each processor's memory. *Global-count* examines a single-bit field in all processors and returns the number of "1" bits. *Global-add* sums multi-bit fields. *Global-max* and *global-min* return the largest (smallest) value found in a specified field across all selected processors. *Global-add* operates on unsigned integers, signed integers, or floating point values, as do *global-max* and *global-min*. The *enumerate* instruction places a different consecutive integer into each of a selected set of processors.

7.4 Communications Instructions

The simplest form of communication between Connection Machine processors is between nearest neighbors. Each processor is wired to its neighbors to the North, East, West, and South by a communications network called the *NEWS grid*. Four instructions, *get-from-north*, *get-from-east*, *get-from-west*, and *get-from-south* control the transfer of data. Information is passed one bit at a time.

General intercommunication and dynamic reconfiguration is performed by a much more

powerful communications system, the Connection Machine *router*. It allows full messages to be sent from any processor to any other; the sending processor simply needs to have the address of the destination processor. Messages may be of any length. Typical messages contain 32 bits of information; adding the address information and headers results in a transmitted package of 50 to 60 bits (depending on the number of virtual processors being used).

Each of the 65,536 physical processors is connected to 16 other physical processors in a special organization (a 16-dimensional hypercube) that provides large numbers of direct paths to distant parts of the system. Every processor is connected to 16 other processors, namely those whose binary address is different in just one of the 16 bits. The following example shows the interconnections of processors 6_{10} and 2070_{10}. The binary addresses are shown in parentheses.

```
    2 ( 0000  0000  0000  0010 )
    4 ( 0000  0000  0000  0100 )
    6 ( 0000  0000  0000  0110 )
    7 ( 0000  0000  0000  0111 )
   14 ( 0000  0000  0000  1110 )
   22 ( 0000  0000  0001  0110 )
   38 ( 0000  0000  0010  0110 )
   70 ( 0000  0000  0100  0110 )
  134 ( 0000  0000  1000  0110 )
  262 ( 0000  0001  0000  0110 )
  518 ( 0000  0010  0000  0110 )
 1030 ( 0000  0100  0000  0110 )
 2054 ( 0000  1000  0000  0110 )
 4102 ( 0001  0000  0000  0110 )
 8198 ( 0010  0000  0000  0110 )
16390 ( 0100  0000  0000  0110 )
32774 ( 1000  0000  0000  0110 )


   22 ( 0000  0000  0001  0110 )
 2054 ( 0000  1000  0000  0110 )
 2066 ( 0000  1000  0001  0010 )
 2068 ( 0000  1000  0001  0100 )
 2070 ( 0000  1000  0001  0110 )
 2071 ( 0000  1000  0001  0111 )
 2078 ( 0000  1000  0001  1110 )
```

```
 2102 ( 0000  1000  0011  0110 )
 2134 ( 0000  1000  0101  0110 )
 2198 ( 0000  1000  1001  0110 )
 2326 ( 0000  1001  0001  0110 )
 2582 ( 0000  1010  0001  0110 )
 3094 ( 0000  1100  0001  0110 )
 6166 ( 0001  1000  0001  0110 )
10262 ( 0010  1000  0001  0110 )
18454 ( 0100  1000  0001  0110 )
34838 ( 1000  1000  0001  0110 )
```

These two sets of addresses have a common connection. Processors 6 and 2070 both connect to 22. Thus it is possible to pass a message, for example, from processor 14 to processor 10262 in just four steps. The router at processor 14 passes it to the router at processor 6, which passes it to 22. From there it goes to 2070 and then to 10262.

7.5 The Routing Process

Connection Machine physical processors are grouped sixteen to a chip. There is a single router on each chip that services all sixteen processors. Hence four of the sixteen routing connections are internal to an individual chip. It takes a maximum of twelve steps to move from any chip to any other chip. During message routing, the system goes through all twelve steps. If the router on a given chip has a message whose relative address has a "1" in the low order bit position, it sends that message on the first of the twelve steps to the chip whose address differs in that same bit (i.e., the next chip). If the message it has has a "0" in the low order relative address bit, the on-chip router does not send any data on that step. The process continues through all twelve steps, with all router chips responding in the same way.

The basic message passing instruction is *send*. Arguments to *send* specify the length of the message and two memory fields. Within each processor, one field contains the message data, and the other contains the address of a destination processor. *Send* causes all active processors to initiate message transfers at once. The special Connection Machine routing hardware handles the volume of messages efficiently. An individual router on a chip may receive as many as twelve messages from other chips during a message cycle, one from each other chip that it is connected to. It can in turn send as many as twelve messages, one on each of the wires. If two messages need to go down the same wire, one is buffered until the next routing cycle. If an individual router becomes extremely busy, it can defer acceptance of any new messages from its own processors. Deferral keeps the router free to

handle messages from other chips. If the chip's buffer space still fills, it refers messages to neighboring chips.

Simultaneous message sending introduces the possibility that the same location in the same processor will receive two or more messages in the same cycle. The simple *send* instruction gives unpredictable results in this case. Several variations of the *send* instruction, such as *send-with-add*, deal with this possibility. If two or more *send-with-add* messages arrive at the same destination, they are summed. *Send-with-overwrite* causes one message to be delivered intact, discarding all other messages directed to that destination. Other options include *send-with-max* and *send-with-logior*.

7.6 Dynamic Reconfiguration

A processor address is all it takes to establish a link on the system. This flexibility allows applications to reconfigure dynamically. A number of intructions support this capability. The *my-address* instruction allows processors to determine their own addresses, so they can send them to other processors and thus establish new connections. The *processor-cons* instruction allows each selected processor to find another "free" processor.

Processor-cons specifies the address of a one-bit field, the "free flag." A processor is considered free if it has a "1" in that field. The system looks in parallel for processors with 1's and passes to each selected processor the address of a different free processor, and at the same time clears the free flags of those free processors.

Chapter 8

Looking to the Future

At one level this report is about algorithms for data level parallel computers: algorithms for looking at the whole problem at once. But at a deeper and more important level, it is really the story of what happened when three very creative people teamed up with a new style of computer, the Connection Machine system. All three people saw new ways to break through old barriers. The computer allowed them to confirm their intuition quickly and then to build upon that intuition.

The intuitive insight behind the document retrieval algorithm is the fact that documents contain a rich set of synonyms for their main content topics. Comparing whole documents could eliminate the need to play guessing games with key words. The idea had never been effectively tested because no conventional computer could execute the algorithms quickly on large data bases. In fact, the first tests on document retrieval by whole document comparison were not particularly encouraging. They were run on a data base of 150 documents, which turned out to be inadequate. When the test was widened to 1500 documents, results were more encouraging. At the level of 15,000 documents, they were outstanding. Without a data level parallel computer such as the Connection Machine system, there would have been no way to even try the approach with 15,000 documents. Test runs would have taken days. Interaction would have been impossible. Now that it has been shown that the algorithm works, whole new possibilities for data base system design are opening up.

The intuitive insight behind the fluid flow algorithm is the fact the behavior of fluids can be simulated without extensive arithmetic computations. Modeling the primitive behavior of molecule packets on a large enough scale can elicit the same macroscopic behavior as real fluids. Tests on the Connection Machine computer suggest strongly that it does. The result is a new and potentially important avenue of scientific investigation.

The intuitive insight behind the contour mapping algorithm is the fact that sophisticated image processing and vision algorithms can be tested on large amounts of data with a small amount of programming effort. The drawing of contour maps, for example, is greatly

simplified by data level parallelism, because it is not necessary to identify the contours one by one and then traverse the perimeter of each one sequentially; instead, each pixel of the contour map "draws itself" in parallel with all the other pixels. Instead of having to break up each phase of the problem into smaller pieces for sequencing purposes, the programmer can tackle it all at once. The result is smaller and simpler programs.

The revolution in data level parallel computing is here. The three algorithms described in this report are only a beginning. But they make an important point: innovative users are an integral part of the story. Users who are stimulated to look at old problems in new ways. Users who revisit problems given up on as impossible in the 60's and 70's. Users who know that a simpler solution is a better solution. These are the users who will assure that the future belongs to computers that look at the whole problem at once.

Bibliography

[1] David C. Blair and M. E. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Comm. ACM*, 28(3):289–267, March 1985.

[2] John F. Canny. *Finding Lines and Edges in Images.* AI Memo 720, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1983.

[3] Michael Drumheller and Tomaso Poggio. On parallel stereo. In *International Conference on Robotics and Automation*, IEEE, April 1986.

[4] U. Frisch, B. Hasslacher, and Y. Pomeau. *A Lattice Gas Automaton for the Navier-Stokes equation.* Preprint LA-UR-85-3503, Los Alamos, 1985.

[5] W. Eric L. Grimson. *From Images to Surface.* MIT Press, Cambridge, Massachusetts, 1981.

[6] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[7] J. Hardy, O. de Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: transport properties and time correlation functions. *Phys. Rev.*, A13(1949), 1976.

[8] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachusetts, 1985.

[9] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[10] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[11] David Marr. *Vision.* W. H. Freeman, San Francisco, 1982.

[12] David Marr and Ellen Hildreth. Theory of edge detection. *Proc. Roy. Soc. London*, B(207):187–217, 1980.

[13] K. Prazdny. Detection of binocular disparities. *Biological Cybernetics*, 52:93–99, 1985.

[14] James B. Salem and Stephen Wolfram. *Thermodynamics and Hydrodynamics with Cellular Automata.* Internal technical report, Thinking Machines Corporation, Cambridge, Massachusetts, November 1985.

[15] David S. Touretzky. *Lisp: A Gentle Introduction to Symbolic Computation.* Harper & Row, New York, 1984.

[16] Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp.* Addison-Wesley, Reading, Massachusetts, second edition, 1984.

Thinking Machines Corporation

Corporate Profile

Thinking Machines Corporation was founded in 1983 based on a belief that developments in and interaction between two rapidly evolving technologies would create an opportunity to design a leading-edge computer system for the future. Parallel processing and artificial intelligence technology are the driving forces which combine to make possible computer systems—hardware and software—with fundamentally new and different capabilities.

Sheryl L. Handler
President and Founder

Ph.D., MIT.

Prior to founding Thinking Machines Corporation, Sheryl was President for 12 years of PACE/CRUX, a domestic and international economic development firm. Clients ranged from biotechnology and telecommunication companies to the World Bank, the U.S. State Department, the U.N. and numerous other agencies and companies. Her experience in working across scientific, engineering and business disciplines provides the basis for assembling and managing the world-class team behind Thinking Machines.

Sheryl was educated at Case Western Reserve, Harvard and MIT, where she was a Collamore-Rogers Scholar.

Richard J. Clayton
Vice President
Connection Machine® Operations

M.S.E.E., MIT.

During his 20 year career at Digital Equipment Corp., Dick was Vice President of Computer Systems Development and Vice President of Advanced Manufacturing Technology. His responsibilities included management of the original VAX hardware design effort. Prior to that he was Group Product Line Manager of Medium Systems.

He has focused the engineering efforts on sound design and ease of manufacturing. Dick is establishing the production, field, and support organization to assure customer satisfaction during the company's accelerated growth.

Marvin Denicoff
Chief of Project Development
and Founder

M.A., Mexico City College, Mexico.

For over 30 years, Marvin was the architect of U.S. government programs in artificial intelligence and related research. As Director of the Computer Science program at the Office of Naval Research, he directed DOD's largest basic research program in artificial intelligence, man-machine systems, and advanced software. His many honors and awards include the DOD/Navy Distinguished Civilian Service Award.

As Chief of Project Development and a founder, Marvin structures joint projects between Thinking Machines and corporations, government agencies, and universities.

W. Daniel Hillis
Founding Scientist

M.S., MIT.

Danny is an acknowledged industry leader in massively parallel systems design. He has made important contributions to artificial intelligence applications in the fields of common sense reasoning and robotics, as well as in the field of systems architecture. He is a former Hertz fellow and author of the book *The Connection Machine.*

Danny is the architect of the Connection Machine system. The design of the hardware is a direct outgrowth of his pioneering work in parallel algorithms and software.

Mirza Mehdi
Vice President for Corporate Development

M.B.A., Georgia State University.

Mirza's experience and accomplishments span the range from major multi-national firms to vigorous young companies. After honing his analytical skills as a Certified Public Accountant with Arthur Andersen & Co., he managed the Business Planning Department of the International Division of Baxter-Travenol Laboratories, where his responsibilities included planning and evaluation of potential acquisitions. Prior to joining Thinking Machines, Mirza was the Director of Finance and Business Development for Genetics Institute, Inc., a major biotechnology firm.

At Thinking Machines, Mirza provides expertise in strategic planning, project evaluation and financial management, and an overview that bridges the scientific and business activities of the company.

Marvin Minsky
Founding Scientist

Donner Professor of Science, MIT

Ph.D., Princeton.

Marvin is a father of the artificial intelligence field and one of its most influential leaders. His work has emphasized approaches to problems of symbolic description, knowledge representation, semantics, machine perception and learning and, recently, in psychological and physiological theories of imagery, memory, and new computational structures. Also an experienced engineer, Marvin was one of the most influential initiators of the modern field of intelligence-based mechanical robots. Among his many honors and awards, he is a Member of the National Academy of Sciences, a Fellow of the American Academy of Arts and Sciences, and the winner of the Turing Award of the Association for Computing Machinery.

At Thinking Machines, Marvin serves as scientific advisor for the company's artificial intelligence projects.

James Bailey
Director of Marketing

B.A., Brown University.

During his 16 year career at Digital Equipment Corporation, Jim managed the New Products Marketing department, which introduced major new corporate products, including the VAX. He has also held positions of corporate manager of pricing, competitive analysis, and market research. He is a member of Phi Beta Kappa.

Jim is responsible for introducing Thinking Machines products and technology to customers, and for establishing joint development programs where appropriate to assure a successful bridge between our technology and the customer application.

Rolf D. Fiebrich
Director of Advanced Systems Development

Dr.rer.nat (Ph.D.) in Computer Science, Technical University of Munich.

During Rolf's career at IBM he was a Research Staff Member of the Thomas J. Watson Research Center, where he focused on research programs for methods of VLSI design. Prior to that he was Assistant Professor of Computer Science at Ludwig-Maximilians—University of Munich. He has published many articles in the field of software engineering and VLSI design.

Rolf manages the company's effort for investigating design advances for future generations of the Connection Machine system. He is responsible for the development of engineering applications that exploit the power of the Connection Machine system. Rolf also manages the company's VLSI design automation development and the custom VLSI program that supports the Connection Machine engineering effort.

Jill P. Mesirov
Senior Scientist

Ph.D., Brandeis University.

Before arriving at Thinking Machines, Jill was the Associate Executive Director of the American Mathematical Society. She still serves as Executive Director of the International Congress of Mathematicians 1986. Previous to that she was a member of the Research Staff of the Communications Research Division of the Institute for Defense Analyses. Her research there focused on the areas of efficient algorithms, cryptology, and speech. She has also published in the field of nonlinear partial differential equations.

Jill leads the scientific computing group at Thinking Machines. Her group is exploring numerical and nonnumerical algorithms for the Connection Machine with a view towards scientific and engineering applications. One exciting project involves the use of cellular automata methods for modeling fluid flow.

George Robertson
Senior Scientist

M.S., Carnegie-Mellon University.

Before joining Thinking Machines, George was Senior Scientist at Bolt, Beranek, and Newman, and Research Computer Scientist in Artificial Intelligence at Carnegie-Mellon. He has done research in distributed systems, multiprocessor systems, programming languages, and user interfaces. He was the principal designer of the large scale ZOG decision support system for the USS Carl Vinson nuclear aircraft carrier.

At Thinking Machines, George is involved in a wide range of research projects, including machine learning, genetic algorithms and classifier architectures, the design of the TMC Indexer interface and advanced testing strategies for the Connection Machine system.

Guy L. Steele Jr.
Senior Scientist

Ph.D., MIT.

Prior to joining Thinking Machines, Guy was Assistant Professor of Computer Science at Carnegie-Mellon University, where he engaged in research in VLSI design, computer architectures, and high-level languages. Guy is the author of *Common Lisp: The Language*, the standard text on the subject. He is also a co-author (with Samuel P. Harbison) of *C: A Reference Manual.* He was Program Chairman for the 1984 ACM Symposium on LISP and Functional Programming, and has served on the program committees of other ACM symposia.

Guy is responsible for the Architecture of the Connection Machine system software and languages. He is furthermore responsible for leading the implementation of CM Lisp™.

Theodore F. Tabloski Jr.
Director of System Software Development

Ph.D., Purdue University.

Ted has had a fifteen year career in industry with responsibilities for product planning, design, and development of advanced computer hardware and software systems. His most recent affiliation has been with Computer Consoles, Inc., where, as Director of Engineering, he participated in the development of corporate strategies and the formulation of marketing plans. Prior to that, Ted was at Bell Laboratories where he held various supervisory and research engineering positions.

Ted manages the system software effort for the Connection Machine program. He is focusing on the development, quality, release and support activities associated with the system software.

David L. Waltz
Senior Scientist

Ph.D., MIT.

Dave joined Thinking Machines from the University of Illinois, Urbana, where he was Professor of Electrical Engineering and Research Professor at the Coordinated Science Laboratory. Prior to that he was a post-doctoral researcher at the Artificial Intelligence Lab at MIT. He was the Editor for AI of the *Communications of the ACM* and Executive editor of *Cognitive Science.* He has published widely in the field of artificial intelligence.

Dave leads the knowledge representation and natural language group at Thinking Machines. Product development efforts are focused on software that accepts unformatted English language data bases and provides convenient access and ultimately question-answering abilities for the data bases.

Corporate Fellows

Richard P. Feynman
Richard Chace Tolman Professor of Theoretical Physics, Caltech

Ph.D., Princeton University.

During his 35 years as a Professor at Caltech, Richard has received the highest honors in his field. He received the Albert Einstein Award from Princeton in 1954 and The A.E.C./E.O Lawrence Award in 1962. He was elected a Foreign Member of the Royal Society in 1965, and received the Nobel Prize in Physics the same year. He received the Oersted Medal for Teaching in 1972 and the Niels Bohr International Gold Medal in 1973.

Numerical algorithms and methods of computation have been specialties of Richard's throughout his career. He has been in the forefront of Thinking Machines' work in developing new algorithms for computing in parallel. His analysis of message traffic and routing algorithms played a central role in the final design of the Connection Machine system.

Charles Leiserson
Associate Professor MIT

Ph.D., Carnegie-Mellon University.

As a graduate student at Carnegie-Mellon, Charles wrote the first paper on systolic architectures with H.T. Kung, for which they received a U.S. patent. His dissertation, *Area-Efficient VLSI Computation,* won the first ACM Doctoral Dissertation Award. In 1981, he joined the faculty of the theory of computation group in the MIT Laboratory for Computer Science. Honored by a Presidential Young Investigator Award in 1985, he is also a member of the IEEE and the ACM, and serves on the ACM General Technical Achievement Award Committee which selects the Turing Award winner. He has authored over twenty papers on the theory of VLSI and parallel algorithms.

At Thinking Machines, Charles applies his expertise on parallel computation, VLSI architectures, graph theory, digital circuit timing, analysis of algorithms, computer-aided design, placement and routing, wafer-scale integration and layout compaction, to the Connection Machine design and applications effort.

Tomaso A. Poggio
Professor
MIT

Doctor in Physics, University of Genoa.

Tomaso is Professor at the MIT Artificial Intelligence Laboratory with an appointment in the Whitaker College of Health Sciences and Technology. He has published widely in the field of computational vision and is on the Editorial Board of seven specialized journals.

Tomaso plays a leadership role in the vision research now underway at Thinking Machines. Among his special interests is stereo vision, a capability which is being applied to the problems of satellite image analysis.

Jacob T. Schwartz
Professor
New York University

Ph.D., Yale University.

Jack is Professor of Mathematics and Computer Science at New York University and former Chairman of the Computer Science Department at the Courant Institute of Mathematical Sciences. He is a member of the National Academy of Sciences, the Editorial Board Chairman of the *Journal of Computer & Systems Sciences,* and the former Chairman of the Computer Science Board—National Research Council. He currently directs the New York University Robotics and Computer Vision Laboratory.

As a founder of the Ultracomputer project at the Courant Institute, Jack is a leading figure in parallel system design. At Thinking Machines he is actively involved in algorithms for massively parallel computation.

Jerome Wiesner
President Emeritus and Institute Professor
MIT

Ph.D., University of Michigan.

Jerome Wiesner is currently teaching and involved with research at MIT. He has been a member of the MIT Faculty for over 25 years, having served as Dean of the School of Science and Provost before becoming President in 1971. He was Science Advisor to Presidents Kennedy and Johnson. Jerome serves on many corporate boards and is an advisor to numerous national and international agencies.

At Thinking Machines, Jerome's role is that of scientific advisor for the company's basic and applied research projects.

Stephen Wolfram
Institute for Advanced Study
Princeton

Ph.D., Caltech.

An internationally known theoretical scientist, Stephen has contributed to a wide range of scientific and computational fields. He is the acknowledged leader in the burgeoning field of cellular automata, a field in which he has published widely. Among his many honors is the MacArthur Foundation Prize.

The Connection Machine system is the first computer system that can fully exploit the computational power of cellular automata. Stephen's work at Thinking Machines has centered on the use of these algorithms for the simulation of physical phenomena such as fluid dynamics and heat transfer.

Board of Directors

Richard J. Clayton

James Cohen
Kassko Enterprises

Marvin Denicoff

Sheryl Handler

W. Daniel Hillis

William McCowen
Hoenig & Co., Inc.

William S. Paley
Founder Chairman, CBS.
Chairman, Executive Committee,
CBS.

Frank Stanton
President Emeritus, CBS.
Former Chairman,
The Rand Corporation.
Overseer, Harvard College.

®Connection Machine is a registered trademark of Thinking Machines Corporation.
™C* and *Lisp are trademarks of Thinking Machines Corporation.
©1986 Thinking Machines Corporation

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Imagine a computer that expands your vision

by looking at the whole problem at once.

Imagine a 1000 Mips computer that is simple to program. That's right, 1000 million instructions per second. And simple to program.

A computer that solves your problems by working on all of the data at once, instead of one word at a time.

A computer that handles words and pictures as easily as numbers.

A computer that is opening new frontiers in applications.

Touch the front cover panel and
65,536 squares respond all at once
just as the processors of the
Connection Machine® system
reconfigure and compute all at once.

Imagine a 1000 Mips computer that is simple to program. That's right. 1000 million instructions per second. And simple to program.

A computer that solves your problems by working on all of the data at once, instead of one word at a time.

A computer that handles words and pictures as easily as numbers.

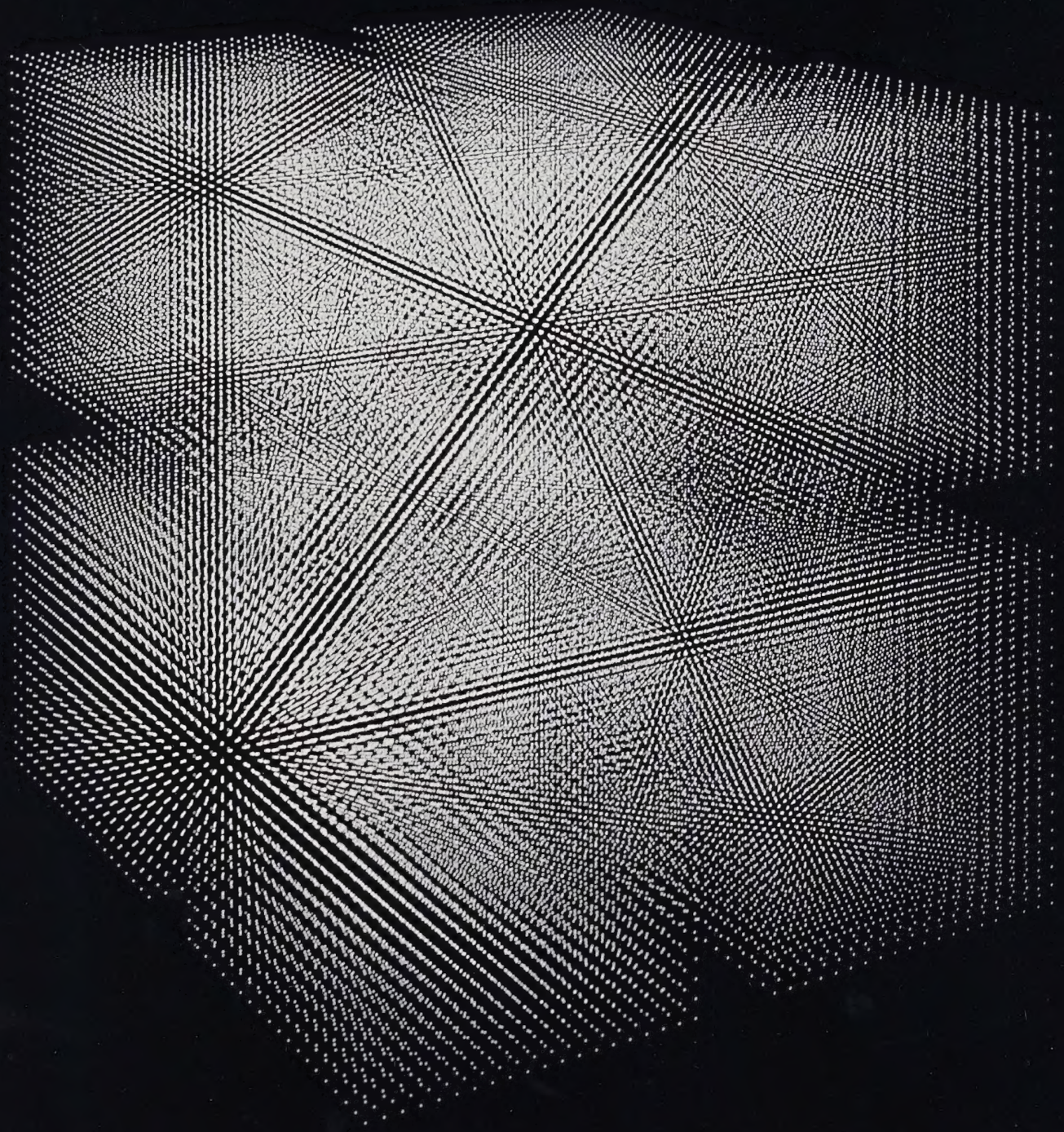
A computer that is opening new frontiers in applications.

The Connection Machine® computer system is here. Reversing a twenty-year trend toward greater and greater complexity in large scale computing systems. Uniting immense processing power with simplicity of programming. Establishing a new standard of large scale computing capability.

The Connection Machine system is easy to use because it has enough computing elements to assign one to each piece of data. Its 64,000 processors scan whole data bases, process whole pictorial images, and simulate whole VLSI circuits simultaneously. All processors are linked together in a dynamic network that reconfigures to match the way the problem works. The result is a simpler to program computer, one that delivers its enormous computing power more naturally and directly.

What do we mean by "simple to program"? We mean that many of your most complicated and time-consuming subroutines can be replaced by single operations that are executed directly by the machine. Not just regular operations like vector arithmetic, but also operations that require complex patterns of data transfer, like network propagation and sorting.

We also mean that it is easy to use. You interact with the familiar programming environment of a front end system, running the editors, utilities, and operating system to which you are already accustomed. You write your program in a simple extension of a high level programming language, like C or Lisp. The compiler and the Connection Machine hardware automatically take care of how the data is allocated to the processors, how operations are synchronized, and what data is communicated from one processor to another. It's good at those things, so you can concentrate on the application, not the computer.



Looking at the whole problem at once means computing on every element of a data structure at the same time. The Connection Machine system's tens of thousands of processors allow it to do this in a straightforward way: by attaching a separate processor to each element of a data structure.

The type of data structure depends on the application. In many language processing applications, data level parallelism means a processor for every word or every meaning. For data base applications, it means a processor for every document. In a numeric simulation, it may mean a processor for every element of a matrix. And in image processing, data level parallelism means a processor for every picture element, or pixel, in an image.

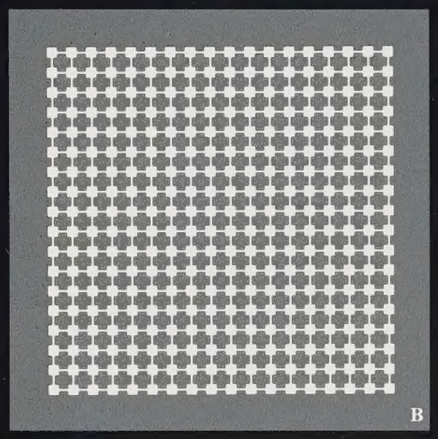
The performance advantages of data level parallelism are dramatic. The parallelism inherent in the data structures grows with the size of the data. Processing all the pixels in an image, for example, is as fast as processing one pixel, because they can all be computed at the same time. A whole data base can be searched in the time it takes to search one document. When computing the flow of air over an airplane wing, the Connection Machine system calculates the flow over all parts of the wing simultaneously, just like it works in reality. Speedups of 1000 or more are common with data level parallelism.

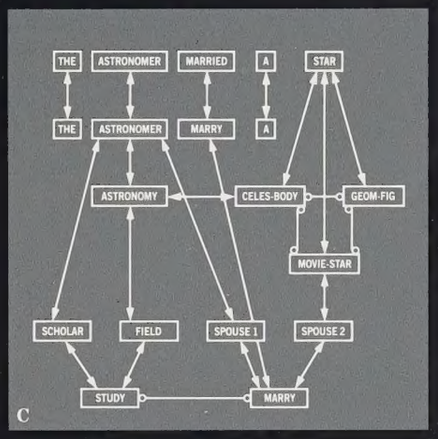
If data level parallelism is so simple and powerful, why don't all big computers work this way? Because in order to make these programs work, tens of thousands of processors have to work together. They have to communicate. The Connection Machine system solves the communications problem in hardware, so the natural algorithms work well.

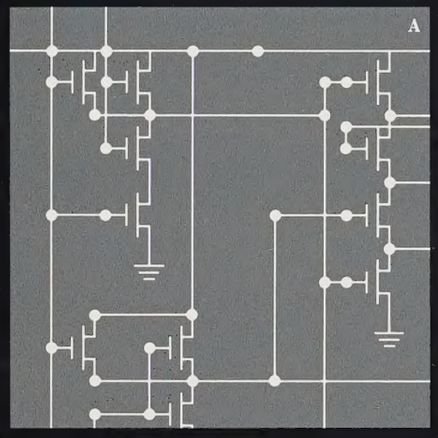
What makes the Connection Machine system work are the connections. Inside the machine there is a very flexible high-bandwidth communication network that moves data between processors at billions of bits per second. Routing circuits on every chip automatically steer data along the fastest paths, helping to make programming simpler. There is no need to adapt your application to the structure of a fixed architecture like a grid, ring, hypercube, or tree. Instead, the Connection Machine system adapts to your application, by dynamically forming the connections that are needed.

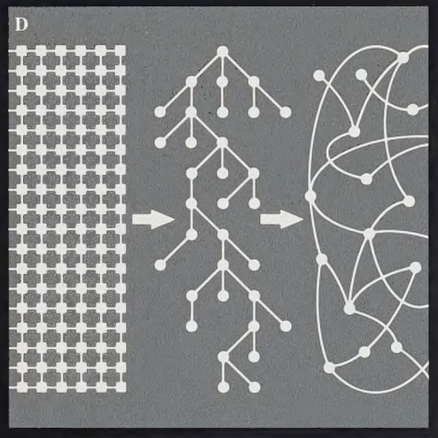
(A) In the simulation of a VLSI integrated circuit, the individual problem element is the transistor. Transistors connect together by their circuit wiring, which is unique to the individual circuit. Some transistors are wired to the transistor next door, some are wired to transistors clear across the chip, and others are wired to both. For every VLSI circuit it simulates, the Connection Machine system reconfigures its internal processor connections to match the wiring of the circuit.

(B) In picture processing, the individual problem element is the pixel. Pixels interact in a grid pattern, with the state of each pixel influencing those adjacent to it. For image processing, the Connection Machine system configures itself as a grid, allowing each processor to pass information to the processors above, below, and to the sides.









(C) In language processing, the individual problem element is the word or the meaning. Words and meanings connect together in increasingly complex structures called semantic networks. The Connection Machine system sets up a connection between each related word and meaning.

(D) The most demanding applications have no fixed topology. They change during the course of the problem. So does the Connection Machine system. A complete vision system may, for example, start its analysis by filtering the image in a grid topology. Global operations, such as computing the average intensity, are naturally tree-like in nature. Bringing together features into objects requires irregular patterns of communication. Only the Connection Machine system adapts dynamically to these changing patterns.

In short, the Connection Machine system's simplicity is also the source of its great power. Large numbers of processors plus a dynamically reconfigurable intercommunications network combine to allow processing to go on directly at the data level, where thousand-fold parallelism is natural.

Connection Machine simplicity and ease of programming extend across the full range of applications where the amount of data is large. In numeric and symbolic applications alike, the system's ability to dynamically configure itself to the natural structure of each problem makes algorithms more natural and implementation more rapid.

Example Application: Words. The Connection Machine system is allowing certain artificial intelligence algorithms to be applied to real-world problems for the first time. One of the most important of these applications is data base search.

The Connection Machine data base system retrieves information by whole document comparison instead of by individual "key word." The user can point to a single relevant document and say, in effect, "Find me all the documents on the same subject as this one." The system compares content-bearing words and phrases throughout the documents. Up to 64,000 whole document comparisons are performed simultaneously.

Whole document comparisons consistently find relevant documents that keyword-based systems miss, because they do not depend on matching any single word. Within a document there are scores of content-bearing words. Many will appear in other documents on the same subject, assuring that an overall match is found even though particular words may be missing. The second advantage of whole document comparison is that it can present the best documents first. Documents with the most significant matches are selected for display at the top of the list.

The Connection Machine retrieval system also eliminates manual indexing of unstructured text. Even huge volumes of incoming text are analyzed automatically by Thinking Machines indexing algorithms, which pick out the content-bearing words and phrases. These phrases form the basis for the subsequent search decisions.

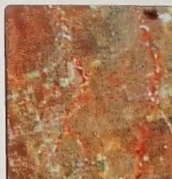
"Whole document" algorithms are just one of the areas of natural language computing that the Connection Machine system is making easy to implement. Interestingly, many of these approaches have been known for 20 years, but until now they have been too difficult to program.

Example Application: Pictures. Today, to gain useful information from satellite images, users must analyze pictorial information directly, and in very high volume. Modern analysis of food production, land terrain, and weather patterns all depend on these pictures.

The Connection Machine system has made it possible to automatically generate contour maps in a few seconds instead of a few hours. The algorithms to do this utilize two images of the same terrain taken from slightly different angles.

After the noise in the images is removed, they are "slid" over each other and minute features compared. Thinking Machines comparison algorithms note the alignment of features at each stage. The more a feature is displaced between the two images, the closer it is to the camera and hence the higher it is in physical terrain terms. This information allows the system to draw and display a contour map of the terrain.

The Connection Machine architecture is ideal for the whole range of vision algorithms and applications. Region labeling algorithms, for example, pick out areas of consistent intensity within a picture and mark them as single objects. Other algorithms pick out additional significant features, such as lines and arcs, for further analysis. The combination of these capabilities, many of which operate at a thousand times the speed of conventional computers, makes the Connection Machine system the computer of choice for demanding vision applications.



Example Application: Numbers. The Connection Machine system brings a new level of power and naturalness to numeric computing applications. The simulation of a VLSI circuit shows how general communication and massive computing power combine to provide an easy-to-program solution. The problem is to feed a wave-form into a circuit, then compute the wave-form that the circuit will produce at any other selected point. For a circuit with tens of thousands of transistors, all switching on and off in parallel, the required number of computations is immense.

The Connection Machine system configures its communications pattern to match each individual circuit exactly. A complete processor is assigned to each transistor and node. The linkages between processors are configured to match the wires between transistors. The simulation matches the reality: all processors (transistors) compute (change state) in parallel.

Processors evaluate mathematical equations that describe the behavior of the device they are simulating. Then they communicate their voltage, conductance, current, and charge to other devices. Finally, the system iterates to reach consistent values for all devices for that time step of the simulation.

VLSI simulation is just one of the two-dimensional and three-dimensional numeric problems that the Connection Machine system is solving. Other examples include matrix algebra and linear programming.

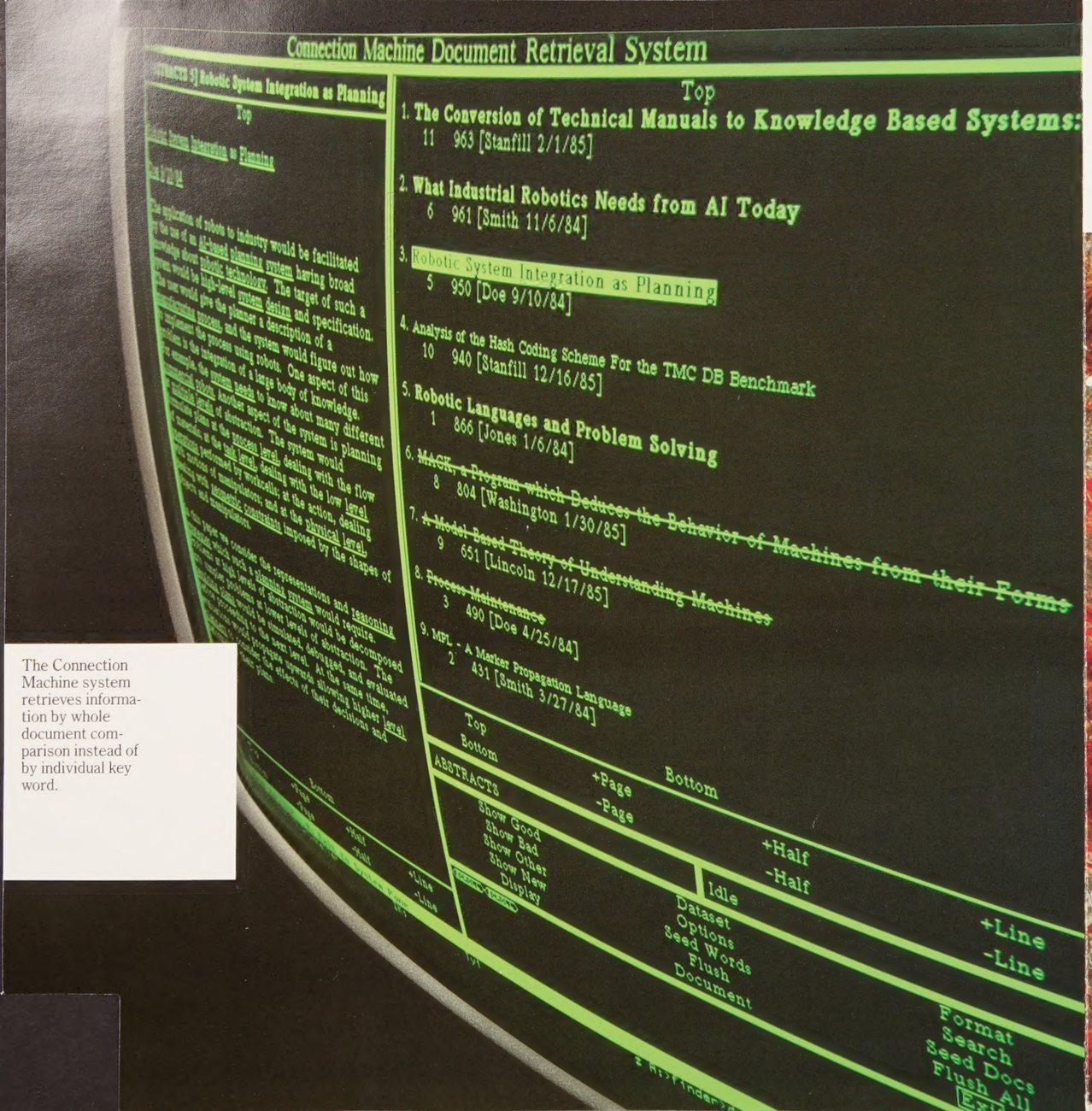
Example Application: Aggregate Behavior. Powerful though they are, words, numbers, and pictures are simply abstractions of the world around us. They simplify behavior that is too complex to deal with directly.

Fluid dynamics is an outstanding example of how the power of the Connection Machine system allows innovative users to cut through these abstractions and reach the physical behavior directly. By analyzing the way fluids flow, scientists design more efficient aircraft and make more accurate weather forecasts. Traditional methods have modeled fluids by partial differential equations. The Connection Machine system provides new insights by more directly modeling the behavior of actual molecules.

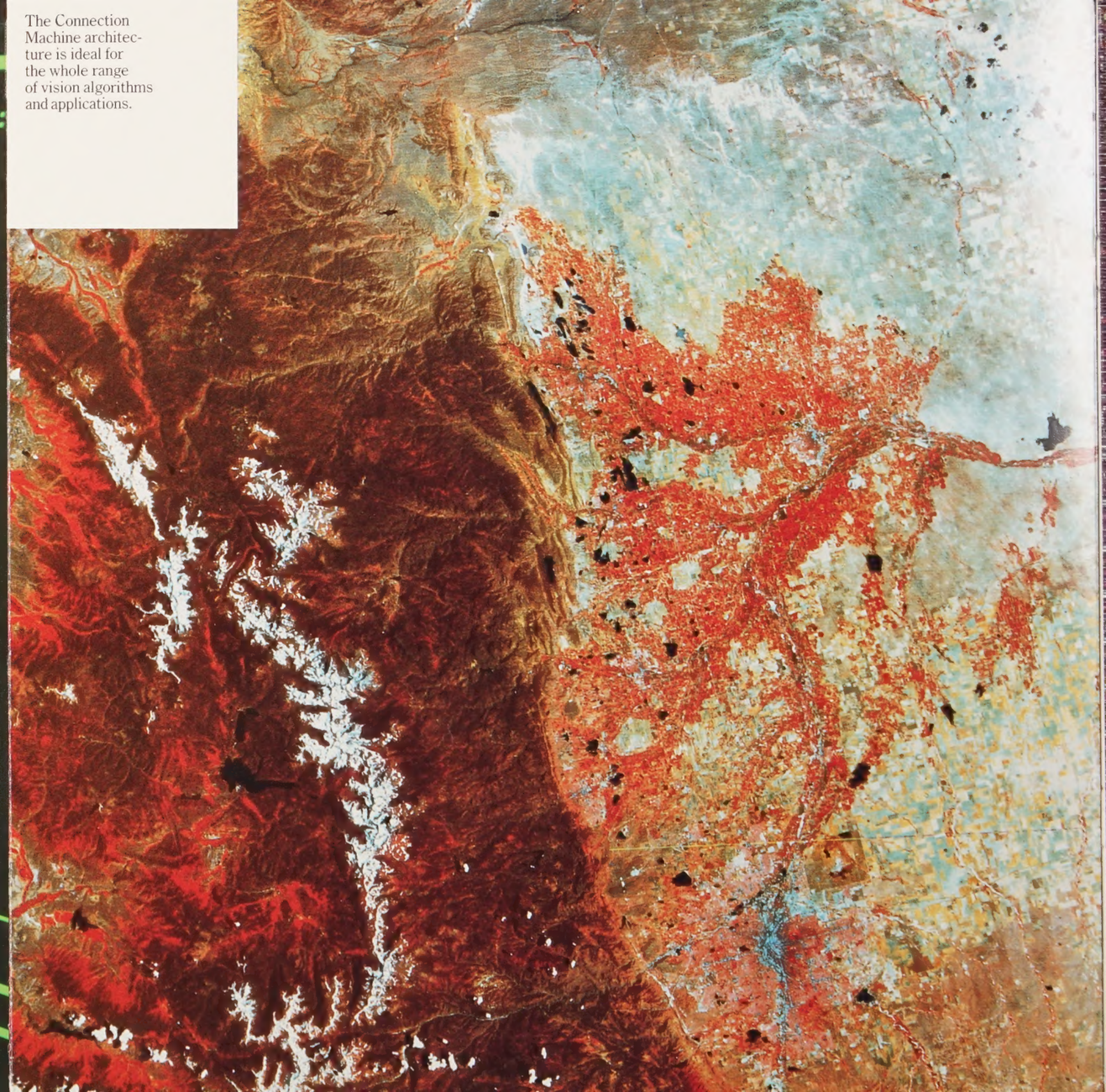
Huge numbers of packets are introduced into the system, each like a tiny bundle of molecules. They move and jostle according to very simple logical rules. No arithmetic of any kind is involved, so the programming is simpler than traditional methods.

As these individual local interactions evolve in huge numbers (the Connection Machine system can update a billion of these individual states per second), the behavior of fluids emerges. Injected from an edge, a fluid flows in a regular way until an obstruction (such as an airfoil) is encountered. Swirls and eddies appear. Over time they lose their energy and trail off.

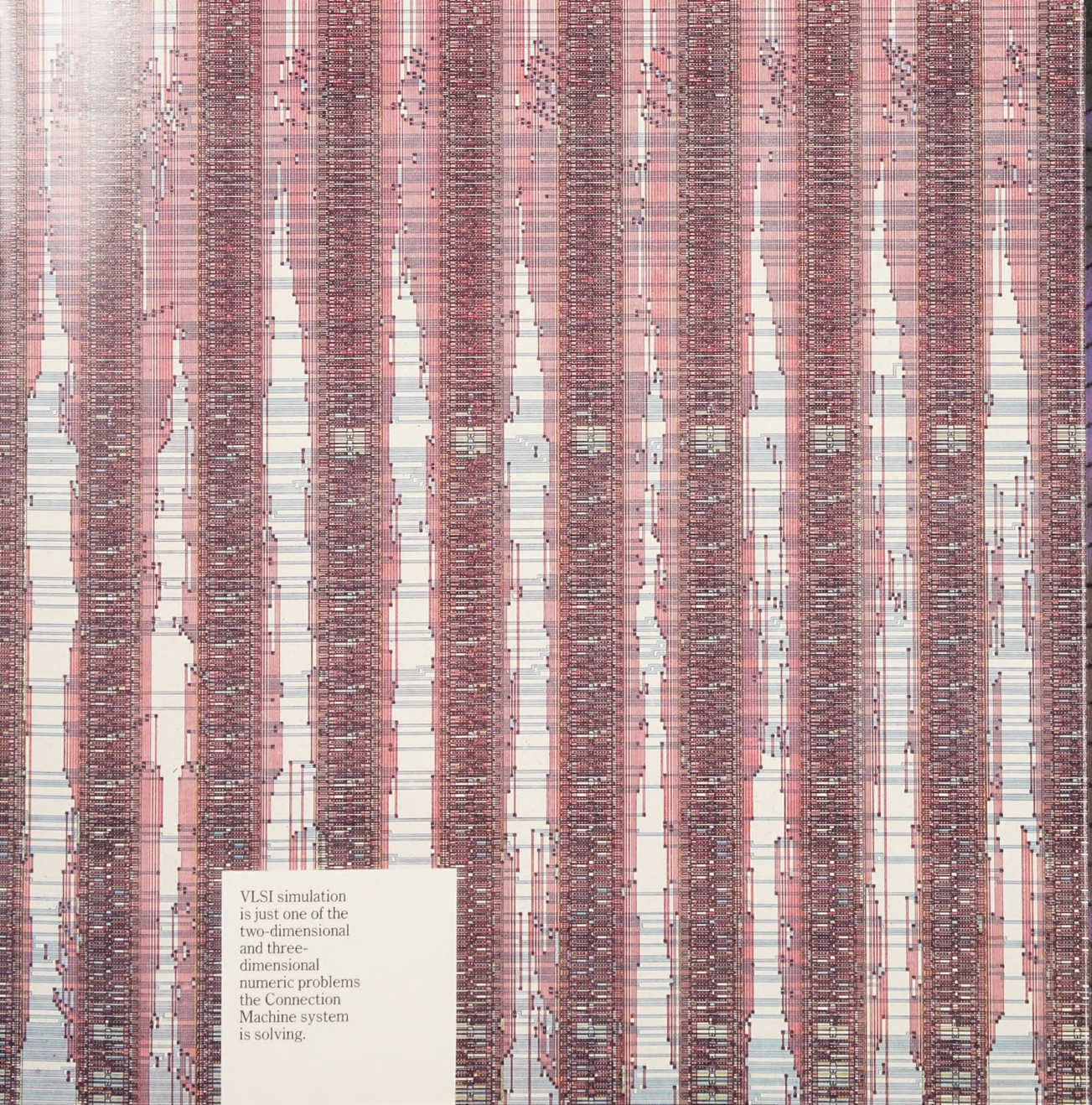
The simplicity of the model makes simulation an interactive process for the first time. A change in the geometry of the object requires no lengthy reinitialization. Results for a new shape are available in a minute or two.



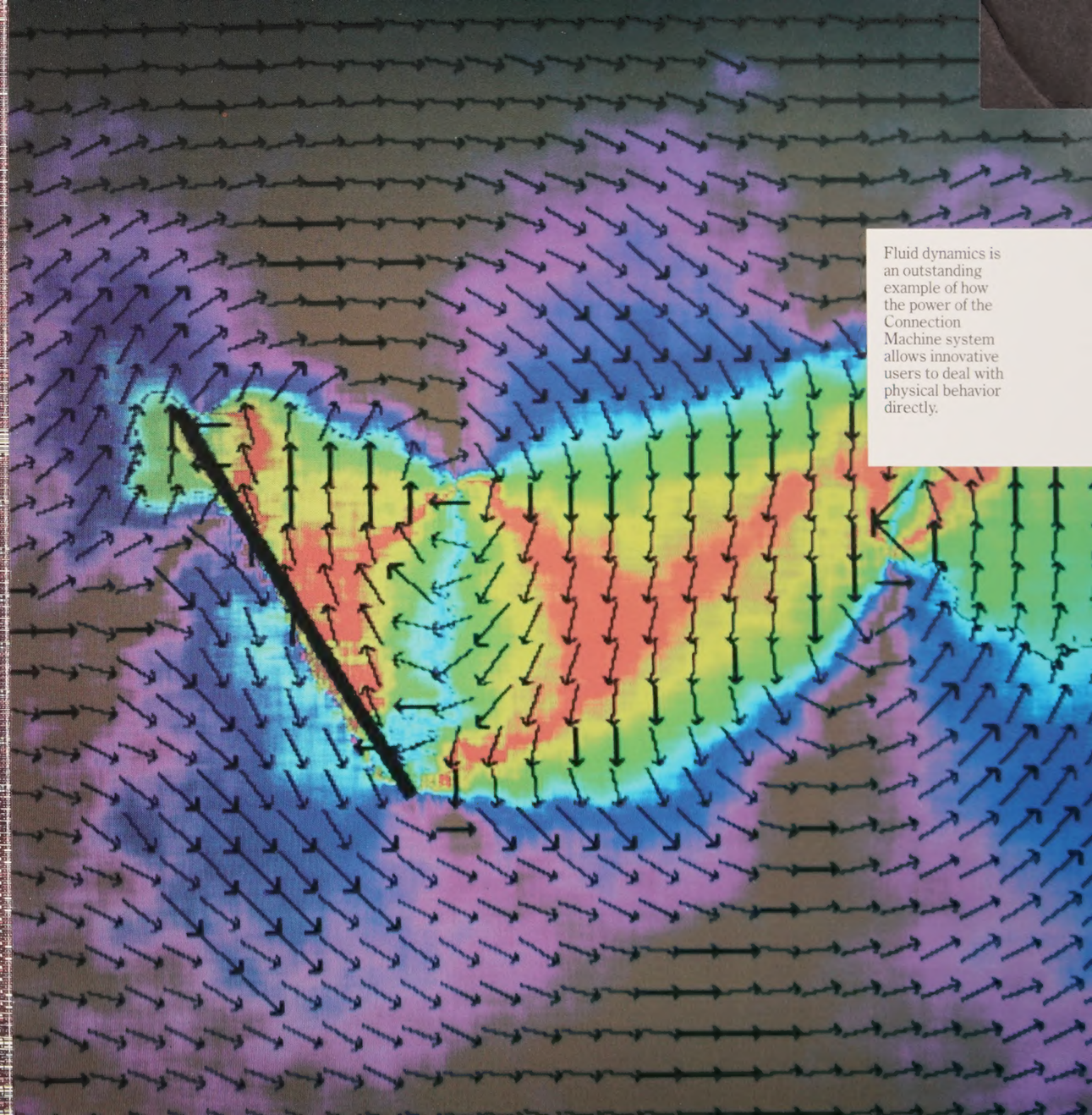
The Connection Machine system retrieves information by whole document comparison instead of by individual key word.



The Connection Machine architecture is ideal for the whole range of vision algorithms and applications.



VLSI simulation is just one of the two-dimensional and three-dimensional numeric problems the Connection Machine system is solving.



Fluid dynamics is an outstanding example of how the power of the Connection Machine system allows innovative users to deal with physical behavior directly.

Programming and Using the Connection Machine System. We no longer have to accept complexity as the price of increased performance in high end computers. The Connection Machine system achieves performance without tricky compilers, ultra-fast components, or exotic packaging.

Simple Construction. The Connection Machine system uses a few simple parts over and over again. A special processor chip implements sixteen processors. 4096 of these chips are used in the system as a whole. Design rules throughout the system are conservative. There are no exotic technologies to cause reliability problems. The system is entirely air cooled. No unusual site preparation is required.

If a failure does occur, fault diagnosis is rapid. Each processor needs only to check itself. 65,536 processors do this testing in parallel. Most failures are isolated to the chip level within five minutes. Hardware is repaired by replacing modules. There are only seven kinds of modules in the system, so a complete set of spares is easily maintained at every site.

Familiar Operating Environment. The user interacts with the Connection Machine system through a conventional front end system, such as a VAX[1] or a Symbolics 3600.[2] The front end supports the operating environment. Installation of a Connection Machine system does not mean introducing a new operating environment into your data center.

The system is programmed via the front end, using familiar editors and utilities. File structures and network protocols are supported there as well, as are the full range of standard VAX and Symbolics peripherals.

A Closer Look: Programming at the Data Level. Computers operate by having a program (the control sequence) operating on a set of data elements to solve a particular problem. Systems that use "control level parallelism" apply their processors to individual sections of the program. The Connection Machine system applies its parallelism directly to the data. The amount of parallelism that can be exploited grows with the size of the data. There is no need to segment the program.

If the data structures contain fewer than 65,536 elements apiece, the Connection Machine system makes the assignment directly. If there are more, the system operates in virtual processor mode, simulating a larger number of processors in a way that is transparent to the user's program. The system can easily support up to 1,000,000 processors in the virtual processor mode.

Variable word length programming and dynamic reconfiguration complete the task of matching the program's data exactly. Data in a Connection Machine system may be as small as one bit or as large as thousands of bits. For picture processing, one- to eight-bit values are common. For numeric processing, 16- to 64-bit words appear most frequently. Language processing values, such as words and sentences, can vary from a few bits to thousands. The Connection Machine system handles them all with equal efficiency.

Dynamic reconfiguration matches the data's connectivity as well as the data itself. When the data structures are set up, so are the linkages. Any processor can send data to any other processor. Within the Connection Machine language environment, data connections are carried out automatically. At the hardware level, the system supports interconnection with a communications system called the router. All 65,536 processors can exchange data simultaneously.

The C* Programming Language. C* is a direct implementation of the data level computing philosophy for general purpose computing. The language is so similar to C that it requires no separate compiler. C* is implemented simply as a preprocessor to the standard C compiler. Consistent with the tenets of data level parallelism, there are no changes to the basic command structure. The control sequence of a C program does not change within C*. Enhancements are at the data level, allowing data structures to be connected to individual processors for rapid execution.

The *Lisp Programming Language. *Lisp is a direct implementation of the data level philosophy for artificial intelligence applications. *Lisp adds data structures to allow direct control of memory allocation and assignment of data values to processors. It gives the programmer direct access to, and control over, the Connection Machine hardware, but does so from within the Lisp programming language. Thus the *Lisp user retains all the productivity benefits of the Lisp machine environment.

The Connection Machine system uses a few simple parts over and over again. A special processor chip implements sixteen processors. 4096 of these chips are used in the system as a whole.

The Future Belongs to Computers That Look At the Whole Problem At Once.

The Connection Machine system operates on a problem's entire data set at once. It is a strikingly simpler approach that opens up whole new possibilities for problem-solving. For picture and image processing, it provides performance levels thousands of times greater than conventional machines. For language processing, it provides access to information in the way people can most easily use it. For scientific processing, it provides access to physical phenomena never before captured in computer simulation.

Yet the impact of data level computing is only just beginning. As innovative scientists are becoming conversant with the Connection Machine system's potential, they are inventing new algorithms and approaches to problem-solving. They are merging the image, language, and numeric processing capabilities of the system into higher level algorithms. Thinking Machines Corporation itself is at the forefront of this new wave of computing. By using knowledge in all its forms, we see the potential for computing systems far more useful and far more powerful than anything available to us today.

®CONNECTION MACHINE is a registered trademark of Thinking Machines Corporation

™The CUBE OF LIGHTS symbol is a trademark of Thinking Machines Corporation

™VAX is a trademark of Digital Equipment Corp.

™SYMBOLICS 3600 is a trademark of Symbolics, Inc.

Some day we will
build a thinking machine.

It will be a truly
intelligent machine.

One that can see and
hear and speak.

A machine that will
be proud of us.

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142
(617) 876-1111

The Economist
10-16 May 1986

SCIENCE AND TECHNOLOGY

The Connection Machine starts to think in parallel

If you have $3m to spare and a penchant for dashing off designs for integrated circuits, searching databases the size of municipal libraries or modelling images in three dimensions, you should invest in a black, five-foot-high cube called the Connection Machine. Made by a small Boston company called Thinking Machines, it was launched last week some years after the first rumours of its power began circulating. One breathless customer, Perkin-Elmer (a computer maker in Northwalk, Connecticut), calls it "the most significant single major advance in large-scale computing in several decades".

The Connection Machine consists of 65,536 minute computers (ie, microprocessors with attached memories) all wired together. Conventional computers—using the so-called von Neumann architecture—have one processor fed by a large memory. The data required for each bit of a computation are shipped into the processor, which does what is needed and ships them back out again. This means that the processor's speed and—more crucially—the speed of the links put an upper limit on the processing power of such a machine. That limit has almost been reached.

There have been several attempts to break this bottleneck and build a "non-von" machine. One is to have several processors, each working on different bits of the problem. Many expensive machines now incorporate more than one processor. Even Crays, archetypal von Neumann machines which fill rooms, cost tens of millions of dollars and crunch numbers at prodigious rates, now have several. Companies like Concurrent (a Perkin-Elmer offshoot) and the Oregon-based Sequent specialise in adding performance to cheaper machines in this way.

The Connection Machine takes multiprocessing to its logical extreme: it splits the problem into thousands of parts and deals with them in parallel. One reason for thinking this is a good idea is that human brains do the same. Mr Danny Hillis sketched out the essentials of such a machine in a doctoral thesis at the Massachusetts Institute of Technology. His ideas were made silicon through the efforts of Dr Sheryl Handler, who put together an impressive team of experts in artificial intelligence, computation and circuit design, and then found $16m from venture capitalists for the new company.

Connections as far as the eye can see

Each processor in the machine is tiny: it can handle "words" of only one binary digit (bit) at a time (the new generation of processors handles 32-bit words), and stores only about 4,000 bits of information on a chip. However, this is enough for all sorts of applications, and such small units come cheap. Many kinds of big problem can be analysed by looking at thousands of tiny sub-problems. A good example is the computer modelling of fluid flows, such as air passing over an aircraft wing. It is relatively easy to calculate how each bit of air is affected by the forces acting on it. It takes millions of calculations—but each of them is simple. Von Neumann machines are poorly qualified to deal with this sort of problem efficiently: most of the expensive beast stands idle while its brain works at a furious rate. The Connection Machine was designed with such problems in mind.

The machine has a number of features that make it more sophisticated than previous attempts at parallelism. Most important is its flexibility. People have tried linking processors in rings, lines, grids and hypercubes (cubes which manage to have more than three dimensions). Mr Hillis stresses that he has made all the links in the Connection Machine programmable with software: the machine hooks itself up, along the lines of the packet switching system used in telecommunications, in whichever way is best suited to the problem in hand. Thinking Machines calls this "data-level parallelism", because the links follow the size and shape of the data.

A big criticism of parallel machines is that they will require new software. The Connection Machine proves that this is nonsense: it runs in the familiar programming languages LISP and C. Indeed, it is designed as a sort of fast processing extension to an ordinary mini-computer, like a DEC VAX or a Symbolics LISP machine. Turning a conventional sequential program into something which will run in parallel is done by the software rather than the user. The user is aware only that his mini has magically speeded up.

The Connection Machine is far from being 65,000 times faster than a conventional machine. But its performance is impressive, especially considering its $3m price tag. Mr Hillis reckons that his machine is up to twice as fast at modelling fluid dynamics as the latest Cray—which costs about $10m. And the Cray is tailored to this kind of problem, whereas the Connection Machine, because of its flexible structure, is a polymath.

In one test, which involved searching a database of 10 billion characters (or about

600-years' worth of issues of *The Economist*), the Connection Machine at three minutes was five times faster than a Cray-with-additions.

Fluid dynamics and counting the number of times Shakespeare wrote "zounds" may not appeal to most computer users. But the speed is the important thing: as always, inventive people will find exciting uses for it. Thinking Machines expects to sell first to scientists, but has its eyes on industry and even finance for later markets. The company has launched, along with the computer, four applications: a very-large-scale-integration package for chipmakers, an artificial-intelligence driven text searcher, a fluid modeller and an image processor. So far Perkin-Elmer has bought one machine, American universities three and the Defence Advanced Research Projects Agency (which dishes out government money for American high-tech) two more. The market for computers costing more than $1.5m already exceeds $10 billion a year.

Copyright 1986 *The Economist*.
Reprinted with permission.

Thinking Machines Corporation

245 First Street, Cambridge, MA 02142-1214 617 • 876 • 1111

Reprinted from THE WALL STREET JOURNAL.

THURSDAY, MAY 1, 1986
© 1986 Dow Jones & Company, Inc. All Rights Reserved.

Thinking Machines Unveils a Computer With New Technique

By a WALL STREET JOURNAL *Staff Reporter*

CAMBRIDGE, Mass. – Thinking Machines Corp. unveiled its first product, a computer called the Connection Machine that uses a technique called parallel processing to solve some problems much faster than traditional computers.

The closely held, three-year-old company has attracted widespread attention in the computer world because of reports of the speed and unusual design of its computer. It was the subject of a front-page article in The Wall Street Journal in February.

Thinking Machines announced two models of its computer, one with 65,536 processors, which will sell for $3 million, and one with 16,384 processors, which will cost $1 million. The individual processors work together on problems such as modeling the flow of water molecules past a boat hull or designing the circuits in a semiconductor. Although each processor is small, in combination they can process one billion instructions per second, the company said. The biggest conventional computers process about 40 million instructions per second.

The company, backed by $16 million in venture capital, grew out of work at the Massachusetts Institute of Technology. It said early customers for the machine, which will be available in July, include two MIT laboratories, the Defense Department, Yale University, and Perkin-Elmer Corp., which will use one to study computer vision, artificial intelligence and simulation.

Scientists have worried that parallel processing would prove unworkable because programming would be so difficult. However, Oliver McBryan, a computer scientist with New York University who has worked with the machine, said it "is an order of magnitude easier to program," than previous parallel processors.

DOW JONES REPRINT SERVICE ● P.O. BOX 300 ● PRINCETON, NEW JERSEY 08540
DO NOT MARK REPRINTS ● REPRODUCTIONS NOT PERMITTED

nature

NATURE VOL. 321 8 MAY 1986

US technology

Parallel computer unveiled

Cambridge, Massachusetts

JUST two and a half years after setting out to build the world's first "fifth generation" computer, Thinking Machines Corporation (TMC) of Cambridge, Massachusetts, last week unveiled the Connection Machine (CM), a massively parallel computer with 65,536 individual processors. TMC has already developed some remarkable applications for the new machine but the company consultant, Richard Feynman of California Institute of Technology, says the problems have yet to be dreamed up that will make full use of CM's unique abilities.

While a graduate student at Massachusetts Institute of Technology (MIT) in Cambridge, CM's spiritual father Danny Hillis came up with the blueprint for a data-level parallel processing computer. Using data-level processing leads to a more natural way of thinking about certain problems, says Hillis. For example, organizing a visual image by examining one small section at a time makes less sense than processing all the information in the image simultaneously. Data-level processing works best on problems with large amounts of data, whereas control-level parallel processing machines are most effective when the ratio of program to data is high. But control-level parallelism involves the difficult task of developing algorithms for asynchronous control of operations.

Communication among processors is the key to CM. Locally, this is done by interconnections among neighbouring processors. For general communications and dynamic reconfiguration, CM uses routers to send messages to the 64K processors arranged in a 16-dimensional hypercube at speeds of up to 3,000 megabits per second.

CM consists of a front-end computer, either a VAX or a Symbolics 3600, that sends instructions to the parallel processors. Each processor contains 4K of memory, so a fully configured CM has 32 megabytes of memory. Even with a $3 million price tag, or $1 million for a CM with 16K processors, prospective buyers are lining up to get hold of the new machine. TMC has so far delivered only one CM, to the Defense Advanced Research Projects Agency (DARPA), but in two months machines will be heading to MIT (2), Yale, Perkin Elmer Corporation and a second machine to DARPA.

In 1983, Hillis teamed up with Sheryl Handler to form TMC. Handler wanted to find a product that would provide a "fundamentally new way of computing". The new company quickly brought on board top scientists as consultants: Hillis's adviser Marvin Minsky and Thomas Poggio of MIT, Stephen Wolfram of the Center for Advanced Study at Princeton and Caltech's Feynman. With $16 million in equity capital and a $4.7 million DARPA contract early on, TMC was able to take the risks Handler felt necessary to leapfrog the competition.

A major advantage of CM for new users is its ability to run in familiar operating environments. Extensions to C and LISP computer languages designed by TMC allow immediate access to CM's parallel capabilities. TMC has already come up with applications packages that show off CM's power. A search for key words in a 16,000-document database takes only 30 milliseconds, much faster than conventional searches. Image processing, very large scale integrated (VLSI) circuit design, and fluid flow problems all take advantage of CM by assigning one processor to each element of the problem's dataset. CM can achieve operating speeds up to 7,000 MIPS (million instructions per second) on certain applications, faster than the fastest computer utilizing more conventional architecture.

Although CM's initial capabilities are impressive, Wolfram believes that a new computer language is needed to take full advantage of it. CM is well suited to cellular automata, and Wolfram predicts that almost all problems now being solved using differential equations will be done in future by constructs like cellular automata.

Another plus for CM is its expandability. While traditional computers using principles developed 40 years ago by von Neumann are inherently limited by a single central processing unit, an arbitrarily large number of processors can in theory be linked together using CM's design. TMC scientists have already written simulations of a machine with one million processors that can run on current CM computers.

Hillis says TMC's next goal is to develop a true learning system. In a small way, the interconnecting processors of the CM resemble the interconnections of individual neurones of the brain. Hillis hopes that this type of architecture will encourage the development of learning algorithms that more closely resemble biological processes.

Other groups around the world are working on new computers using parallel architecture, but Poggio points out that CM has one obvious advantage over other projects: "It works". **Joseph Palca**

Reprinted with permission.

Thinking Machines Corporation

245 First Street, Cambridge, MA 02142-1214 617 • 876 • 1111

The Boston Globe THURSDAY, MAY 1, 1986 BUSINESS

Computer firm banks on speed

By Ronald Rosenberg
Globe Staff

Thinking Machines Corp. put some sparkle in the computer industry yesterday with a super-fast computer that it says will change the way traditional computer functions are done.

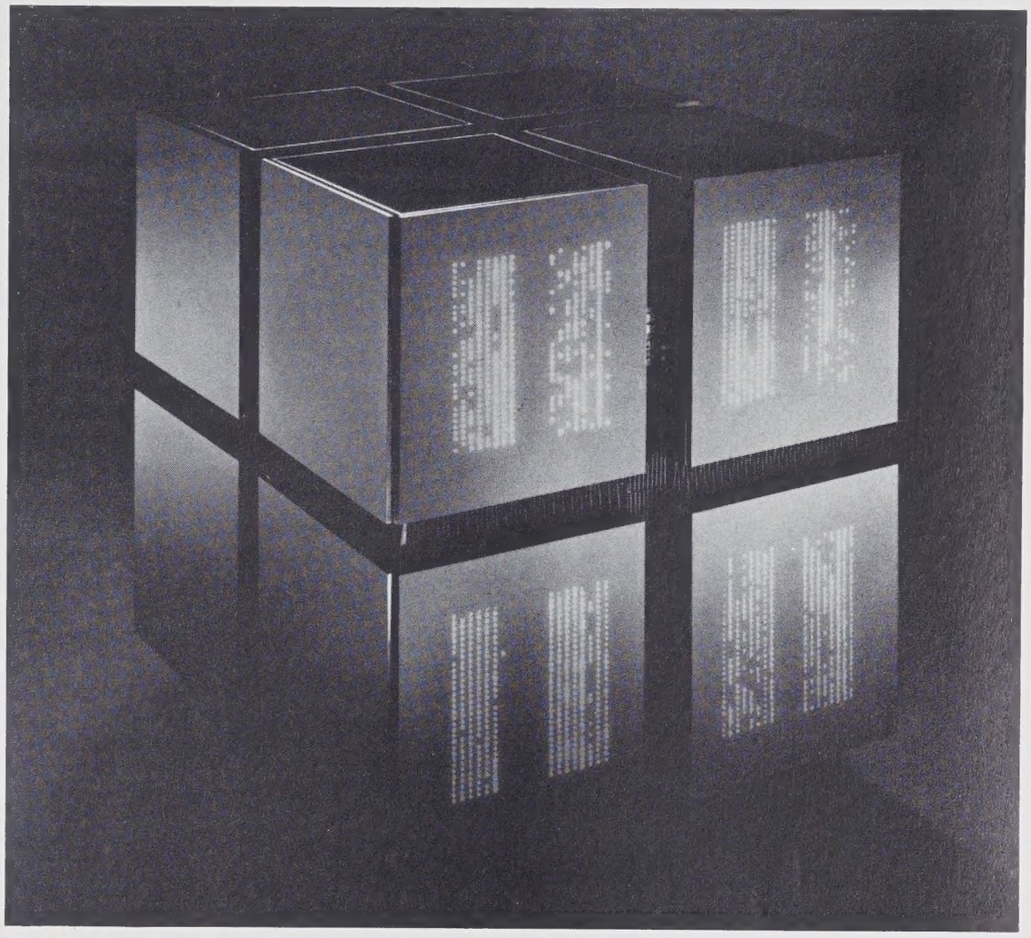
At a press conference in the company's Cambridge headquarters along the banks of the Charles River, company officials led by Sheryl L. Handler, the founder and president, introduced the Connection Machine. It is a computer that can process information at the rate of 1 billion instructions per second. (By comparison, International Business Machines Corp.'s top-of-the-line Sierra mainframe is rated at 28 to 40 million instructions per second).

The high speeds can reduce the search and retrieval of a voluminous database to seconds instead of many minutes. To electrical engineers the Connection Machine promises to speed up the simulation of electronic circuits. Scientists can do complex modeling used in weather forecasting, automobile streamlining and airplane design more expeditiously.

"This is quite an achievement," said Jacob T. Schwartz, a professor of mathematics and computer science at New York University who developed some of the Connection Machines' algorithms as a consultant.

Distinguishing the Connection Machine from other computers is the interconnection of between 16,000 and 64,000 custom-designed microprocessors. Previous machines have used no more than several hundred processors. Unlike conventional computers that process information in a sequence, piece by piece, this computer processes all the data at the same time. The high speed is achieved by connecting the microprocessors, which act on the information and communicate with other microprocessors.

The five-foot-high computer is an imposing machine. It consists of eight black cubes, each with 512 flashing red lights that are visible through a semitransparent surface at one end. Although the lights aid in computer maintenance they are used largely for marketing.



Thinking Machines Corp. new Connection Machine system.

"I love the lights," chuckled W. Daniel Hillis, the 29-year-old architect of the Connection Machine who first explored the concept while a graduate student at Massachusetts Institute of Technology. "I want to know when the computer is working, and looking at the video display terminal is not enough."

His doctoral thesis adviser was Marvin Minsky, considered the father of the artifical-intelligence field, who like Hillis is a founding scientist of the company. Minsky predicted similar machines will be on desktops in 10 years.

Both men, together with Sheryl L. Handler, who founded Thinking Machines in June 1983, expected the research and development to take about five years. She, too, is an MIT graduate and a businesswoman.

"Where we are today is what I expected would take five years," said Handler, who drew some criticism by attracting scientists to leave academia and work for Thinking Machines. "We couldn't have accomplished the Connection Machine if we relied on university research. We needed to bring the best people together."

Thinking Machines also attracted $16 million in capital from such investors as William Paley, the founder of CBS who is also an investor in another Cambridge company, Genetics Institute.

"Having the top scientists impressed me," said Paley, who confers with Handler at least once a month. "Scientists are pretty careful and quiet guys. This field is a lot different from the business and entertainment field."

Critics of the Connection Machine claim its high price of $1 million to $3 million and the need for specially adapted software may limit its impact. Others point to the wave of minisupercomputer machines coming this year that although less powerful are also less costly.

But others contend it will be a winner.

The company has sold six machines, including one each to MIT and Yale University.

Reprinted courtesy of The Boston Globe.

The Boston Herald

Thursday May 1, 1986

New computer closest thing to human brain

By GEOFFREY ROWAN

A COMPUTER which can solve problems in seconds that take traditional computers hours to unravel was unveiled yesterday by a Cambridge company.

The Connection Machine, by Thinking Machines Inc., uses 64,000 separate processors in unison to create a computer which thinks more like a human than a standard machine.

Comparing the speed of traditional computers to the Connection Machine is like "comparing a bicycle to a supersonic jet," said the computer's designer, W. Daniel Hillis.

Its incredible power — the Connection Machine can process 1 billion instructions per second — comes from a technology called parallel processing.

Conventional computers use a single, large processor to compute information in a linear fashion, extracting one piece of data from its memory, processing that and returning to its memory before moving on to the next part of the problem.

The Connection Machine uses 64,000 processors working on different pieces of a problem at the same time, much as the human brain depends upon billions of neurons working at the same time to keep many different human functions going.

"The result is a computer that looks at the whole problem at once," Hillis said.

Other companies have attempted to create parallel processing machines but they have worked on a smaller scale. Most are considered multiple processors because while they use several processors they do not operate in parallel.

Thinking Machines said it will market the $3 million Connection Machine to government agencies, Fortune 500 companies with large research departments and universities.

The first six machines have been purchased by the Perkin-Elmer Corp., the Media Laboratory at the Massachusetts Institute of Technology, the Artificial Intelligence Laboratory at MIT, Yale University and the government's Defense Advanced Research Project Agency (DARPA), which has already received one and has another on order.

"We think the Connection Machine system is the wave of the future," said Prof. Patrick Winston, director of MIT's Artificial Intelligence Laboratory.

"The existence of a machine like this changes the way we think about problems," he said. "It will open up new ways of thinking in many areas."

Reprinted with permission.

Thinking Machines Corporation

245 First Street, Cambridge, MA 02142-1214 617 • 876 • 1111

A 1-BIPS SYSTEM TAKES NEW TACK IN PARALLELISM

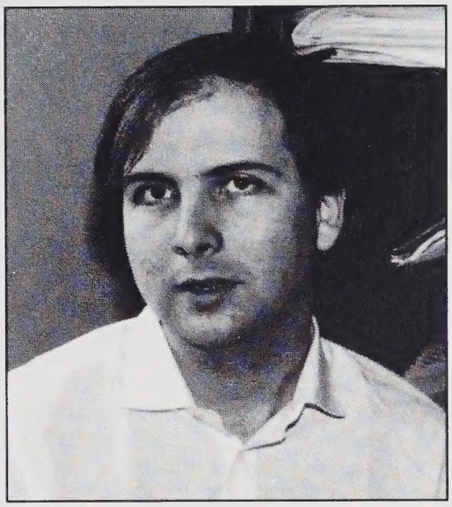
CAMBRIDGE, MASS.

Mips worshipers have a new god. Thinking Machines Corp.'s Connection Machine, a massively parallel computer that made its commercial debut last week, goes beyond millions of instructions per second and ushers in the era of bips with its ability to execute 1 billion instructions/s.

Other numbers describing the Cambridge company's machine are likewise grand in scale. In its largest configuration, the computer has 65,536 processors and 32 megabytes of memory. Processors can communicate with each other at 3,000 Mb/s and the computer's input/output channel can handle 500 Mb/s. The air-cooled computer, housed in a cluster of eight translucent black cubes clustered one on one in four stacks with dozens of blinking red lights, needs a hefty front-end machine, either a Digital Equipment Corp. VAX or Symbolics Inc. computer.

More important than the raw numbers, however, is that the computer takes a new approach to parallelism. The Connection Machine uses data-level parallelism by reducing tasks to data elements and assigning a single processor to each element, says W. Daniel Hillis, founding scientist at Thinking Machines. This makes it particularly useful for applications involving huge amounts of data, as opposed to heavy computing on a few data elements.

As an example, Hillis describes an attempt to determine which of 50,000 news articles relates to a particular clipping. In the Connection Machine, he explains, each article would be assigned to and examined by a single processor. The processors would then compare their data elements with the reference clipping stored in the front-end computer.



HILLIS. The Connection Machine devotes a processor to each data element in a problem.

ALL AT ONCE. "The most rational way to execute a lot of operations is to work all the data at once," says Hillis. This fine-grain parallel-computing method also avoids the primary bugaboo of so-called coarse-grain processors (those with fewer, larger elements), which require subdivision of the application problem, he adds. "We don't partition, and that avoids a big set of problems."

The Connection Machine's cornerstone is a custom chip that holds 16 processors and 4-K of memory. If an application takes only some processors, the system temporarily switches off unused chips. If there are more data elements than processors, the computer's hardware operates in virtual-processor mode by subdividing memory and simulating additional multiple processors, each with a smaller memory. The Connection Machine can support up to a million virtual processors. The machine's breakthrough concept was eliminating the separation of processors and memory and instead mixing them together along with high-speed communication elements, says president Sheryl Handler.

The Connection Machine runs under conventional operating systems, including AT&T Co.'s Unix, DEC's VMS, and Symbolics's Lisp environment. Its languages are extended versions of Lisp and C. Throughout development, says Handler, the development team concentrated on applications. At the large conference announcing availability of the machine, demonstration areas included document processing, contour mapping, chip design, and fluid dynamics.

The company also announced that several machines had already been sold, in addition to one delivered last fall to the Defense Advanced Research Projects Agency, which partly funded development of the computer and has placed an order for another machine. Other early customers are Perkin-Elmer Corp., the Massachusetts Institute of Technology, and Yale University.

Perkin-Elmer's 16,000-processor unit will be used at MRJ Inc., Oakton, Va., to search text, process images, and do research, says Edward McMahon, a member of the Perkin-Elmer division's technical staff. Referring to the difficulty of programming other multiprocessors, McMahon says, "The creativity in using the Connection Machine is in formulating algorithms, rather than the programming part."

Price for the computer with 65,536 processors and 32 megabytes of memory is $3 million. A unit with 16,384 processors and 8 megabytes of memory sells for $1 million. This compares favorably with large mainframes, claims vice president Richard Clayton. He calculates the Connection Machine's cost at $3,000 per mips or less, whereas mainframes cost up to $150,000/mips. To keep costs down, he explains, the company placed a priority on using simple building blocks and conservative manufacturing technology. *–Craig D. Rose*

Reprinted from **ELECTRONICS**, May 5, 1986, copyright 1986 by McGraw-Hill, Inc. with all rights reserved.
Additional reprints may be ordered by calling Electronics Reprint Department (609) 426-5494.

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214

Electronic Engineering
TIMES

Issue 379 The Industry Newspaper For Engineers And Technical Management Monday, April 28, 1986

Thinking Machines' Parallel Processor Connects Over 64,000 Nodes

By R. Colin Johnson

CAMBRIDGE, Mass. — Thinking Machines Corp. will launch its 64,000-node parallel processor here this week.

The success of the Connection Machine System in experimental work sponsored by the Defense Department led to Thinking Machines' release of a commercial version (see *EE Times,* July 29, Page 1).

The Connection Machine is a so-called fine granularity parallel processor, so called because it harnesses upwards of 64,000 nodes in its base configuration. Future versions are envisioned that could use a million or more nodes for supercomputer performance.

The Connection Machine currently has the finest granularity of any commercial parallel processor, though NCR has a new chip from which a system that rivals the Connection Machine in granularity could be built.

NCR's geometric arithmetic parallel processor, GAPP has 72 processors on a CMOS chip.

The Connection Machine System keeps its processors doing useful work with a VLSI interconnection scheme that mirrors the job being run. Nodes and their interconnections are closely matched to particular applications.

Artificial-intelligence researchers have been especially keen on the Connection Machine System as an expert system engine. "We are aiming at using the Connection Machine to retrieve elements of a knowledge base in just a few machine cycles rather than seconds or minutes," said Edward Feigenbaum, the author of *The Fifth Generation* and a DOD reseacher.

The trick is that each node in the Connection Machine System is set up to hold one element of knowledge. Then its connections are configured to mirror the relationship among the knowledge-base elements. Searching for information and unifying knowledge is then perfectly matched by the supporting VLSI.

Of the emerging parallel processing technologies, the Connection Machine System is probably one of the more esoteric. It usually requires special programming techniques that are, at best, little understood. And it makes little attempt to run standard software. But applications that can be adapted to its architecture have been shown to run orders of magnitude faster.

Origins Of The Machine

The Connection Machine is the brainchild of researcher Daniel Hillis, who developed the idea while at MIT. Prototypes of the Connection Machine have generated excitement among other researchers at MIT, who have been using it to experiment with parallel processing architectures.

An advantage of the machine in this type of research is the simplicity with which connection topologies can be set up. "The programmable interconnection greatly simplifies the problems of trying to get serial algorithms to emulate parallel architectures," said Charles Leiserson, a professor at MIT who works with VLSI architectures. "People around here are finding it indispensable for running simulations of architectures," Leiserson said.

Tomaso Poggio, an MIT researcher, claims that his system can reconstruct three-dimensional scenes from two-dimensional information, using only about one quarter of the machine's 64,000 processors.

Other academics have also heaped praise on Hillis' invention. AI expert Marvin Minsky feels the system represents a fundamental break with computer-design tradition. Up to now, all computers have had basically the same architecture—one or a few large memory banks.

Challenging that conception, the Connection Machine links together thousands of millions of extremely small processors and memories. "It may take generations to unfold the implications of this new species of machine," Minsky said. He will be present at the formal unveiling this week.

But the question of how this machine will fare in an increasingly crowded parallel processing market now looms. One promising area that has been rapidly proven in research projects is VLSI design. "The machine's high degree of parallelism and general communications structure seems a natural fit to applications such as logic and circuit simulation, routing and placement, design rule checking, compaction and circuit extraction," Hillis said.

A processor can be allocated to each component in a circuit and the wiring pattern programmed into the Connection Machine's communication pathway to create a high degree of realism in a simulation.

—Additional reporting by Chappell Brown

Thinking Machines Corp.
245 First St., Cambridge, MA 02142-1214; (617) 876-1111

Copyright© 1986 by CMP Publications, Inc., 600 Community Drive, Manhasset, N.Y. 11030
Reprinted with permission from Electronic Engineering Times.

Reprinted from
COMPUTERWORLD

Massively parallel processor introduced

Connection Machine concept represents a 'new technology'

By Eddy Goldberg

CAMBRIDGE, Mass. — The massively parallel processing Connection Machine introduced by Thinking Machines Corp. last week represents "a fundamentally new technology," according to company founder and President Sheryl Handler.

The Connection Machine incorporates 16,000 to 64,000 processors that reportedly process data at up to 1 billion instructions per second.

"I think it's a breakthrough, but I don't think it immediately makes other machines obsolete," said Vincent E. Giuliano, vice-president and chief scientist at Mirror Systems, Inc., which produces software systems for parent company Times Mirror Co.

"The important thing at this point is to prove the concept," said Martin Schultz, chairman of Yale University's computer science department. "We have to find out if massive parallelism is the answer to our needs," he said. Schultz is looking to simulate three-dimensional physical experiments, among other tasks.

The company, founded in June 1983, has sold six units, which cost $1 million in the 16,000-processor configuration and $3 million for the full 64,000-processor version.

The first sale was to the Defense Advanced Research Projects Agency (DARPA), which sponsored the project to develop a general-purpose massively parallel computer.

"Each of the architectures we're sponsoring represents a major architectural advance," said Stephen Squires, assistant director of DARPA's Information Processing Technology Office. "This is pushing things in the direction of fine-grained, massively parallel, fully connected computing," he added.

Fine-grained processing assigns each data element of a task to an individual processor, allowing thousands of processors to work simultaneously, the company said. It works best where the number of data elements is large, ranging from 10,000 to one million.

The front end is a conventional computer, such as a Digital Equipment Corp. VAX or a Symbolics, Inc. 3600, which contains the programs. Any single-data-element instructions are executed directly by the front end. Instructions that call for operations on the whole data set at once are passed to the Connection Machine for execution.

For problems with more than 16,000 or 64,000 data elements, the processors, which each have 4,096 bits of memory, act as virtual processors. The Connection Machine can support up to one million virtual processors.

Its communications technology, called the router, has an overall capacity of 3 billion bit/sec. and allows any processor to establish a link to any other processor in a maximum of 12 dynamically configurable steps. Though its future usefulness is yet to be fully determined, four applications were demonstrated: document retrieval, fluid dynamics modeling, creating contour maps from aerial photographs and the design of very large-scale integrated circuits.

Thinking Machines Corp.
245 First Street
Cambridge, Ma. 02142-1214